

Studienreihe Informatik

Elfriede Fehr

**Semantik von  
Programmiersprachen**

Springer-Verlag

## Vorwort

Dieses Buch entstand als Ausarbeitung des Manuskriptes einer gleichnamigen, vierstündigen Vorlesung, die ich im Sommersemester 1981 an der TH Aachen und in den Wintersemestern 1981/82 sowie 1982/83 an der Universität Bonn gehalten habe. Es sollten die in der theoretischen Informatik bekannten Methoden der Formalisierung der Semantik von Programmiersprachen behandelt werden, wobei der Standardsemantik ein besonderes Gewicht eingeräumt wurde. In der Fachliteratur existieren voneinander unabhängige Darstellungen der zugrundeliegenden mathematischen Theorie, der technischen Methoden und der Anwendungen. Es gibt nur wenige Arbeiten, in denen verschiedene Methoden der Definition der Semantik von Programmiersprachen zusammengestellt und verglichen werden, etwa von Riedewald et al., Donahue, Hoare und Lauer, Pagan oder den Übersichtsartikel von Marcotty et al. Ziel dieses Buches ist es, einerseits in das gesamte Gebiet der formalen Semantik von Programmiersprachen einheitlich und systematisch einzuführen und andererseits die Standardsemantik mit ihrem mathematischen Hintergrund umfassend zu behandeln.

Als besonders geeignete Quelle für die Behandlung der Standardsemantik hat sich das bekannte Buch von Gordon erwiesen, das vor allem in Kapitel 4 eingegangen ist. Neben den im Literaturverzeichnis angegebenen Schriften habe ich insbesondere zur Stoffauswahl auch Vorlesungsmanuskripte von Klaus Indermark, Robin Milner und Christopher Wadsworth verwendet, die sie mir freundlicherweise zur Verfügung stellten.

Mein Dank geht an die Hörer der Vorlesungen, die mich durch ihr Interesse zum Schreiben dieses Buches ermuntert haben. Die drei Tutoren Thomas Boehmsdorff, Johann Tamme und Sigrid Warhaut haben durch viele kritische Bemerkungen das Manuskript mitgestaltet. Besonderer Dank für wertvolle Anmerkungen gebührt den Herausgebern Wilfried Brauer und Gerhard Goos sowie dem anonymen Gutachter des Springer-Verlags. Für gründliches Korrekturlesen danke ich Eva Pless, Sigrid Warhaut und Anne Fehr.

Die äußere Form des Manuskriptes ist mit Hilfe des von Knuth entwickelten Textsystems  $\text{\TeX}$  und der darauf aufbauenden Weiterentwicklung  $\text{\LaTeX}$  von Lamport entstanden.

Bonn, im November 1988

Elfriede Fehr

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
<b>2 Verschiedene Methoden der formalen Semantikspezifikation</b>	<b>9</b>
2.1 Die Beispielsprache WHILE . . . . .	9
2.2 Informelle Beschreibung der Semantik von WHILE . . . . .	13
2.2.1 Diskussion der informellen Semantik von WHILE . . . . .	15
2.3 Operationelle Semantik der Sprache WHILE . . . . .	16
2.3.1 Die WSKEA-Maschine . . . . .	17
2.3.2 Diskussion der operationellen Semantik von WHILE . . . . .	24
2.3.3 Die Reduktionssemantik der Sprache WHILE . . . . .	25
2.3.4 Äquivalenz von operationeller und Reduktionssemantik . . . . .	27
2.4 Denotationelle Semantik der Sprache WHILE . . . . .	38
2.4.1 Äquivalenz von operationeller und denotationeller Semantik	44
2.5 Axiomatische Semantik der Sprache WHILE . . . . .	47
2.5.1 Beziehung der axiomatischen zur denotationellen und operationellen Semantik . . . . .	53
<b>3 Mathematische Grundlagen</b>	<b>56</b>
3.1 Theorie der semantischen Bereiche . . . . .	56
3.1.1 Elementare Bereiche . . . . .	63
3.1.2 Kartesische Produkte und Folgen . . . . .	64
3.1.3 Summen . . . . .	67
3.1.4 Funktionen . . . . .	69
3.1.5 Rekursiv definierte Bereiche . . . . .	72
3.2 Der getypte $\lambda$ -Kalkül als Metasprache . . . . .	73
3.3 Lösung rekursiver Bereichsgleichungen . . . . .	87

<b>4 Detaillierte Behandlung der denotationellen Semantik</b>	<b>97</b>
4.1 Spezielle Funktionen und Konventionen . . . . .	97
4.1.1 Curry-Isomorphismen . . . . .	98
4.1.2 Die bedingte Verzweigung . . . . .	99
4.1.3 Basisoperationen . . . . .	101
4.1.4 Rekursion . . . . .	101
4.1.5 Modifikation von Funktionen . . . . .	103
4.1.6 Die verallgemeinerte Komposition . . . . .	104
4.2 Denotationelle Semantik der Sprache WHILE unter Verwendung der neuen Notationen . . . . .	105
4.3 Entwicklung der Standardsemantik unter besonderer Berücksichti- gung der Fortsetzungssemantik . . . . .	109
4.3.1 Fortsetzungen . . . . .	110
4.3.2 Fortsetzungssemantik der Sprache WHILE . . . . .	111
4.3.3 Typüberprüfung . . . . .	114
4.3.4 Modifikation des Ausgabebereiches . . . . .	115
4.3.5 Modifikation des Bereiches SPEICHER . . . . .	116
4.3.6 Standardwertebereiche . . . . .	117
4.3.7 Deklarationen . . . . .	118
4.3.8 Standardfortsetzungsfunktionen . . . . .	119
4.3.9 Fortsetzungstransformationen . . . . .	120
4.3.10 Verallgemeinerung der Wertzuweisung . . . . .	121
4.3.11 Standardsemantik von Prozeduren und Funktionen . . . . .	122
4.4 Die Standardsemantik der Sprache PASCAL <sub>0</sub> . . . . .	124
4.4.1 Syntax von PASCAL <sub>0</sub> . . . . .	124
4.4.2 Semantik von PASCAL <sub>0</sub> . . . . .	125
4.4.3 Bemerkungen zur Definition von PASCAL <sub>0</sub> . . . . .	127
4.4.4 Berechnung der denotationellen Semantik eines Beispielpro- gramms . . . . .	130
4.5 Weitere Sprachkonzepte, analysiert im Rahmen der Standardsemantik	133
4.5.1 Sprünge und Abbrüche . . . . .	133
4.5.2 Verallgemeinerungen des Prozedur- und Funktionskonzeptes	136
4.5.3 Datenstrukturen . . . . .	141

<i>Inhaltsverzeichnis</i>	IX
<b>5 Funktionale Programmiersprachen</b>	<b>151</b>
5.1 Die Programmiersprache LISP . . . . .	152
5.1.1 Syntax von Kern-LISP . . . . .	155
5.1.2 Statische Semantik von Kern-LISP . . . . .	157
5.1.3 Operationelle Semantik von Kern-LISP . . . . .	158
5.1.4 Inkonsistenzen von LISP . . . . .	161
5.2 FP-Systeme . . . . .	164
5.3 Programmieren mit rekursiven Gleichungssystemen . . . . .	170
<b>6 Anwendungen der denotationellen Semantik bei der Implementierung höherer Programmiersprachen</b>	<b>177</b>
6.1 Systematische Codeerzeugung aus der Standardsemantik . . . . .	178
6.2 Implementierung nach Übersetzung in kombinatorische Ausdrücke	185
6.3 Implementierung auf Reduktionsmaschinen . . . . .	188
6.3.1 Die Reduktionsmaschine von Berkling und Kluge . . . . .	188
6.3.2 Die Baumarchitektur von Mago . . . . .	189
6.3.3 Die Graphreduktion nach Wadsworth . . . . .	190
<b>Literaturverzeichnis</b>	<b>192</b>

# Einleitung

*Programmiersprachen* sind künstliche Sprachen, die der Kommunikation zwischen Mensch und Maschine dienen.

Beim Studium von Sprachen unterscheidet man allgemein drei Aspekte: Syntax, Semantik und Pragmatik.

- Die *Syntax* beschreibt die lexikalische und die grammatische Struktur einer Sprache unabhängig von ihrer Interpretation.

Die meisten syntaktischen Aspekte einer Programmiersprache werden durch kontextfreie Grammatiken spezifiziert.

Gebräuchliche Techniken dafür sind die Backus-Naur-Form (BNF) [8] oder die Syntaxdiagramme nach Wirth [140].

Programmiersprachen sind jedoch im allgemeinen nicht kontextfrei. Die syntaktische Korrektheit von Programmen hängt auch davon ab, ob die auftretenden Bezeichner konsistent mit ihren gegebenen Deklarationen benutzt werden. Die Formalisierung des kontextabhängigen Teils einer Programmiersprache ist zwar mittels kontextsensitiver Grammatiken möglich, wird aber meist nicht für erforderlich erachtet, zumal dafür ein recht aufwendiger Formalismus nötig wäre. So findet man in Sprachbeschreibungen etwa im Anschluß an die BNF-Regeln einige nicht formal spezifizierte Kontextbedingungen der Art: „Jede Variable muß vor ihrem ersten Auftreten deklariert sein“, oder „Keine Variable darf in einem Block mehrfach deklariert werden“.

Neben den Regeln zum Aufbau syntaktisch korrekter Programme sind auch Algorithmen zur Analyse erforderlich, die ein gegebenes Programm daraufhin prüfen, ob es den Syntaxregeln genügt, und die seine Zerlegung in Komponenten angeben.

Abstrahiert man von der konkreten Darstellung eines Programms und betrachtet nur den Aufbau aus seinen Komponenten, dann spricht man auch von der *abstrakten Syntax* dieses Programms.

Zu Programmiersprachen existieren die gewünschten Analysealgorithmen (siehe z.B. Aho und Ullman [1]), die von der ‘string’-Repräsentation eines Programms den dazugehörigen *Syntaxbaum* erzeugen. Daher kann man bei der Untersuchung der anderen Sprachaspekte davon ausgehen, daß ein Programm in seiner abstrakten Syntax vorliegt. Dies ermöglicht insbesondere

ein direktes Ansprechen der unmittelbaren Programmkomponenten. Die Kontextbedingungen werden im Rahmen des Übersetzerbaus meist bereits zur Semantik gezählt, da ihre Überprüfung in der Phase der Analyse der statischen Semantik erfolgt (siehe z.B. Zima [141]), im eigentlichen Sinne des Syntaxbegriffes gehören sie jedoch zur Syntax.

- Die *Semantik* beschreibt die Interpretation, d.h. den Bedeutungsgehalt einer Sprache.

So können Sätze aus verschiedenen Sprachen die gleiche Semantik haben, z.B. der englische Satz: „I go to school“ und der deutsche Satz: „Ich gehe zur Schule“. In natürlichen Sprachen kann aber auch ein Satz mehrere voneinander verschiedene Interpretationen haben. Betrachtet man den Satz: „Man hat drei Buchstaben“, so ergeben sich mindestens zwei unterschiedliche Interpretationen:

1. Die Semantik von „man“ ist eine Referenz auf das syntaktische Wort „man“, etwa als Lösung der Aufgabe: „Nenne ein Wort mit drei Buchstaben“.
2. Die Semantik von „man“ ist eine Person in einem allgemeinen Sinne, etwa in einem Buchstabenspiel, bei dem an einem gewissen Spielstand jeder Spieler drei Buchstaben hat.

Dieses Beispiel zeigt, daß die Semantik von deutschen Sätzen nicht eindeutig ist und daß die Bedeutung von Wörtern und Sätzen nur im Kontext zu erklären ist.

Seit den frühen sechziger Jahren, als erste Arbeiten über Semantik von Programmiersprachen entstanden, besteht Einigkeit darüber, daß die Bedeutung eines Programms durch seine Wirkung auf den Zustandsraum einer Maschine beschrieben werden kann. McCarthy schreibt: „The meaning of a program is defined by its effect on the state vector“ [84]. Unterschiede gibt es in der Art, wie diese Wirkung formal definiert wird.

Der Wunsch nach einer eindeutigen, präzisen Semantikspezifikation bei Programmiersprachen besteht aus folgenden Gründen:

1. Programme höherer Programmiersprachen müssen erst in Maschinensprache übersetzt werden, bevor sie ausgeführt werden können. Zur *Konstruktion eines geeigneten Übersetzers* (Compilers) benötigt man genaue Kenntnis der Semantik, um die Portabilität von Programmen zu gewährleisten.
2. Bei Vorliegen einer formalen, standardisierten Semantikspezifikation ist eine *automatische Übersetzererzeugung* möglich. Auf Ansätze in dieser Richtung wird in Kapitel 6 genauer eingegangen.
3. Ein Programmierer braucht eine möglichst genaue Kenntnis der Semantik der verwendeten Sprache, um *Sicherheit beim Entwurf von Programmen* zu gewinnen.

4. Bei standardisierten Problem- und Semantikspezifikationen gibt es Ansätze, die eine *automatische bzw. rechnerunterstützte Programmierung* zum Ziele haben.
  5. Zur Formulierung und zum Beweisen von Programmeigenschaften wie Korrektheit, Terminierung und Äquivalenz ist ein formaler Rahmen erforderlich.
- Die *Pragmatik* behandelt den Gebrauch einer Sprache sowie alle Aspekte, die die Beziehung der Sprache zu ihren Benutzern und ihren Empfängern betreffen. Dazu gehören sowohl eine intuitive Begriffsbildung als auch eine einfache Übersetzbarkeit in die jeweiligen Maschinensprachen.  
Z.B. wird die Wahl der konkreten Syntax vorwiegend nach pragmatischen Aspekten getroffen. Landin hat den Begriff „*syntactic sugar*“ zur Bezeichnung von Spracherweiterungen geprägt, die sich semantisch auf bereits vorhandene Konstrukte zurückführen lassen, aber besser lesbar oder einfacher zu handhaben sind als ihre äquivalenten Umschreibungen [67].  
Typische pragmatische Fragestellungen sind auch Fragen nach dem Zweck und der Wirkung eines Satzes bzw. Programmes.  
Im Rahmen einer Sprachbeschreibung gehört etwa folgender Satz zur Pragmatik: „Eine Anweisung der Form  $I := E$  empfiehlt sich, wenn der Wert des Ausdruckes  $E$  im weiteren Programm mehrfach verwendet werden soll“. Die Pragmatik von Programmiersprachen wurde bisher nicht formalisiert, jedoch wären einheitliche Ansätze zur formalen Beschreibung pragmatischer Aspekte sicher wünschenswert.

Dieses Buch beschäftigt sich mit verschiedenen Methoden der formalen Semantikspezifikation höherer Programmiersprachen. Während die syntaktischen Aspekte bereits zufriedenstellend formalisiert sind und weitgehend einheitlich in der Literatur behandelt werden, gibt es bei der Semantikspezifikation ein breites Spektrum unterschiedlicher Ansätze, die sich grob als operationell, denotationell und axiomatisch klassifizieren lassen. Diese Ansätze sollen hier kurz vorgestellt und im nächsten Kapitel exemplarisch behandelt werden.

1. Die *operationelle Methode* beschreibt die Wirkung von Programmen als schrittweise Zustandsänderung einer abstrakten oder konkreten Maschine. In einem engen Sinne ist jeder Übersetzer (Compiler) oder Interpretierer eine operationelle Semantikspezifikation, insoweit implizit folgende Vereinbarung getroffen wird: die Bedeutung eines Programms ist der Effekt, den eine schrittweise Abarbeitung des übersetzten Programms auf den Anfangszustand der Maschine hat. Dabei gehen jedoch viele spezielle, maschinenabhängige Details in die Semantikspezifikation mit ein, die die abstrakte Struktur der Bedeutungsinhalte verwischen. Daher verwendet man bei der operationellen Methode meist abstrakte Maschinen mit wenigen, übersichtlichen Komponenten, die dann von konkreten Maschinen simuliert werden können.

Im Extremfall bildet die Programm-Daten-Kombination selbst den Anfangszustand, der durch systematische syntaktische Transformationen, die nach bestimmten Regeln angewendet werden, in eine Darstellung des Ergebnisses überführt wird. In einem solchen Falle spricht man auch von *Reduktionssemantik*. Eine Spezifikation im Stile der Reduktionssemantik findet vor allem bei funktionalen Sprachen Verwendung. Darauf soll in Kapitel 5 näher eingegangen werden.

Die grundlegende Arbeit auf dem Gebiet der operationellen Semantik wurde von Landin geleistet. Er beschrieb 1964, wie die wesentlichen Sprachstrukturen höherer Programmiersprachen auf natürliche Weise in die  $\lambda$ -Notation von Church übertragen und dann auf einer geeigneten Kellermaschine ausgewertet werden können [67]. Als Beispiel einer Anwendung dieser Ideen sei auf das Buch von Bauer und Wössner [12] hingewiesen, in dem in Abschnitt 1.7 die operationelle Semantik rekursiver Programme, basierend auf einer Kellermaschine, erklärt wird.

Eine Weiterentwicklung der operationellen Semantik, insbesondere zur Behandlung von Sprachen mit Prozedurkonzept, findet sich in den Arbeiten von Langmaack [69] und [70], in denen die *Kopierregelsemantik* begründet wird.

Wenn die Semantik einer Programmiersprache im Hinblick auf eine korrekte Implementierung formalisiert werden soll, so ist sicher die operationelle Methode am besten geeignet. Hingegen ist das Beweisen von Programmeigenschaften bei operationell definierten Sprachen meist recht mühsam. In der Regel lassen sich kaum die mathematischen Eigenschaften der durch ein Programm definierten Ein-/Ausgabefunktion verwenden, sondern das Verhalten der zugrunde liegenden Maschine muß in Induktionen über die Zustandsfolge simuliert werden. Beispiele solcher Beweise findet man bei Wegner [133], [134] und [135], bei McGowan [88] und [89] sowie bei BurSTALL [21].

Die operationelle Methode wurde bei einigen für die Praxis relevanten Programmiersprachen verwendet. Die beiden wichtigsten Beispiele sind vielleicht die Semantikspezifikationen von ALGOL 60 durch Landin 1966 [68] sowie von PL/I durch Lucas und Walk 1970 [78].

Eine formale Beschreibung der operationellen Semantik einer Programmiersprache enthält das Buch von Greibach [41]. Eine allgemeine Methode zur Beschreibung von operationeller Semantik ist die im Wiener IBM-Laboratorium entwickelte Metasprache VDL (*Vienna Definition Language*). In dieser Sprache lassen sich die Strukturen sowohl der abstrakten Syntax als auch der abstrakten Maschinen als markierte Bäume definieren. Die Sprache verfügt ferner über elegante Methoden, geeignete Funktionen zur Manipulation solcher Bäume aufzuschreiben. Eine übersichtliche Beschreibung von VDL findet man bei Wegner [136].

2. Die *denotationelle Methode* abstrahiert von der schrittweisen Zustandsänderung einer Maschine und ordnet in statischer Weise jedem Programm die entsprechende Ein-/Ausgabefunktion als Semantik zu. Dies geschieht induk-

tiv über den Aufbau des Programms, indem jeder Programmkomponente ein geeignetes mathematisches Objekt als Wert zugeordnet wird. Diese Objekte sind im allgemeinen diskrete Objekte, stetige Funktionen oder Funktionale höherer Ordnung, aus denen dann die Semantik des Programms aufgebaut wird. Der entscheidende Gedanke dabei ist, daß Programmschleifen auf der Ebene der Semantikfunktionen durch Fixpunktbildung modelliert werden können; daher spricht man auch von *Fixpunktsemantik*.

Die Grundidee der denotationellen Semantik, nämlich die Bedeutung der elementaren syntaktischen Einheiten direkt in einer allgemein verständlichen Sprache anzugeben und die Bedeutung der zusammengesetzten Einheiten induktiv unter Verwendung der Bedeutung ihrer unmittelbaren Komponenten zu erklären, kann bereits auf Frege [35], Carnap [24] und Tarski [117] zurückgeführt werden. Die ersten Ansätze zur denotationellen Semantik von Programmiersprachen stammen von McCarthy. Er entwickelt eine Methode, die es ermöglicht, einfache Flußdiagramme in rekursive Gleichungssysteme über Zustandsvektoren zu übersetzen [84].

Die wohl wichtigste Grundlage für die heute üblichen denotationellen Spezifikationen wurde von Strachey entwickelt [114]. Allerdings war zunächst nicht klar, ob es mathematische Modelle gäbe, in denen Lösungen der dort verwendeten rekursiven Gleichungen gefunden werden könnten.

Schließlich gelang es Scott 1969, eine geeignete Modellklasse, nämlich bestimmte stetige vollständige Verbände, zu konstruieren ([107] und [108]). Später wurden anstelle von stetigen Verbänden als semantische Bereiche meist kettenvollständige Halbordnungen, *cpo's* (*complete partial order*), gewählt, bei denen die Konstruktionen von Scott ebenfalls möglich sind.

Eine umfassende Darstellung der denotationellen Semantikspezifikation auf der Grundlage des Ansatzes von Scott und Strachey findet man in dem Buch von Stoy [113]. Eine weitere ausführliche Darstellung dieser Theorie enthalten die Bücher von Milne und Strachey [90], die besonders als Nachschlagewerk zu empfehlen sind. Eine kurze Einführung in das Gebiet der denotationellen Semantik bietet die Arbeit von Tennent [118]. Für den Praktiker erscheint das Buch von Gordon [39] gut geeignet, in dem die wichtigsten Techniken der denotationellen Methode vorgestellt werden, jedoch ohne den mathematischen Hintergrund zu erläutern.

Die denotationelle Methode der Formalisierung der Semantik einer Programmiersprache eignet sich besonders gut dazu, die Bedeutungen möglichst abstrakt, kurz, übersichtlich und vollständig anzugeben. Daher gewinnt man wegen der guten Lesbarkeit ein hohes Maß an Sicherheit beim Programmentwurf. Darüber hinaus existieren bei der denotationellen Semantikspezifikation mathematische Methoden zum Beweisen von Programmeigenschaften. Die wichtigsten davon sind die *Berechnungsinduktion* (siehe Manna [80] und Manna et al. [81]), die *strukturelle Induktion* (siehe Burstall [20] und Aubin [5]), die *Rekursionsinduktion* (siehe McCarthy [84]) und die *Fixpunktinduktion* (siehe Park [100]).

Eine ausführliche Diskussion dieser Beweismethoden einschließlich vergleichender Betrachtungen findet man in dem Buch von Manna [80], in den

Arbeiten Manna et al. [81] und Milner [91] sowie in dem Buch von Loeckx und Sieber [76].

Ein wichtiger Anwendungsbereich der denotationellen Semantik ist der Compilerbau. Es existieren mehrere Ansätze, aus einer standardisierten denotationellen Semantikspezifikation automatisch Übersetzer zu erzeugen. Einige davon werden in Kapitel 6 vorgestellt.

Als typisches Beispiel für die denotationelle Semantikspezifikation einer bekannten Programmiersprache, nämlich PASCAL, sei die Arbeit von Tennent [119] genannt.

Eine Variante der denotationellen Methode ist VDM (*Vienna Development Method*), die von Bjørner und Jones beschrieben wird [16]. Mit Hilfe von VDM spezifizieren sie die Semantik der Sprache Ada [17].

Ebenfalls eng verwandt mit der denotationellen Semantik ist die *algebraische Semantik*. Hier wird die Tatsache ausgenutzt, daß die abstrakte Syntax eine freie Algebra  $\tilde{S}$  mit den syntaktischen Konstruktoren als Operationen ist. Nun lassen sich die semantischen Bereiche ebenfalls als Algebren mit der gleichen Operatormenge auffassen. Da es zu jeder solchen semantischen Algebra  $\tilde{A}$  einen eindeutigen Homomorphismus  $h_{\tilde{A}}$  von  $\tilde{S}$  nach  $\tilde{A}$  gibt, läßt sich die Semantik eines Programms  $P$  auf kanonische Weise als den Wert von  $h_{\tilde{A}}$ , angewendet auf  $P$ , in  $\tilde{A}$  definieren.

Eine Einführung in die algebraischen Techniken findet man in dem Buch von Arbib und Manes [4], in Guessarian [42] oder bei Goguen et al. [38].

3. Die *axiomatische Methode* abstrahiert noch einen Schritt weiter. Hier werden keine konkreten Zustände transformiert, sondern nur noch logische Aussagen über Zustände.

Die Wirkung eines Programms  $P$  wird durch eine möglichst schwache Vorbedingung (*weakest precondition*), gegeben durch eine logische Formel  $Q$ , und eine möglichst starke Nachbedingung, gegeben durch eine weitere Formel  $R$ , charakterisiert. Man notiert diesen Zusammenhang in der Klausel

$$\{Q\}P\{R\}$$

und liest:

Wenn die Bedingung  $Q$  für einen Anfangszustand erfüllt ist, dann gilt nach Ausführung von  $P$  die Bedingung  $R$  für den Endzustand.

Die Vorgehensweise der axiomatischen Semantikspezifikation einer Programmiersprache ist nun wie folgt: Jedem atomaren Bestandteil einer Sprache wird ein Axiom, eine elementare Relation zwischen Vor- und Nachbedingungen, zugeordnet. Jedem zusammengesetzten Programmteil wird eine Ableitungsregel zugeordnet, mit deren Hilfe sich aus den Vor- und Nachbedingungen seiner Komponenten die Vor- und Nachbedingung dieses Programmteils gewinnen läßt.

Der erste Ansatz in diese Richtung ist die Arbeit von Floyd [34], bei dem jedoch die Bedingungen den Kanten des jeweiligen Flußdiagramms zugeordnet

werden. Die entscheidende Entwicklung leistete Hoare 1969, indem er die Relationen zwischen Vor- und Nachbedingungen direkt auf den Programmtext zurückführte [48]. Nach ihm heißt die axiomatische Methode auch *Hoare-Semantik* und der entsprechende Kalkül *Hoare-Kalkül*.

Als Beweismethoden stehen alle prädikatenlogischen Beweisverfahren zur Verfügung. Spezielle Techniken gibt es zur Aufstellung von Schleifeninvarianten. Das Buch von Dijkstra [31] enthält zahlreiche Beispiele für eine systematische Entwicklung korrekter Programme unter Verwendung der axiomatischen Semantik. Von Dijkstra stammt auch der Begriff der „weakest precondition“.

Das Buch von Loeckx und Sieber [76] enthält eine gute Einführung in die mathematischen Methoden der Programmverifikation unter Verwendung der axiomatischen Semantik.

Allgemein lässt sich sagen, daß die axiomatische Methode für Verifikationssysteme am leichtesten zugänglich ist, jedoch eine direkte Compilererzeugung nicht erlaubt. In der Regel ist die Formalisierung der Semantik einer Programmiersprache nicht vollständig in dem Sinne, daß jedes korrekte Tripel der Form  $\{Q\}S\{R\}$  auch in dem Hoare-Kalkül hergeleitet werden kann.

Zu einigen höheren Programmiersprachen existieren axiomatische Semantikspezifikationen, so bei Donahue [32] sowie bei Hoare und Wirth [52] zu PASCAL und bei London et al. [77] zu EUCLID.

Einen genauen Überblick über den aktuellen Stand dieser Theorie findet man in der Arbeit von Apt [3]. Eine ausführliche Darstellung der axiomatischen Spezifikation gängiger Programmformen bietet das Buch von de Bakker [10].

Neben den oben erwähnten Methoden der formalen Semantikspezifikation gibt es noch einige Ansätze, sowohl die Syntax als auch die Semantik einer Programmiersprache mit einem Formalismus zu spezifizieren. Solche Ansätze können als Vorstufe zu einer formalen Semantik oder als Brücke zwischen Syntax und Semantik betrachtet werden. Sie sind zwar im Rahmen des Übersetzerbaus als eindeutige Sprachbeschreibung und als Konstruktionshilfe einsetzbar, eignen sich aber weniger als Hilfe beim Programmentwurf und bei der Programmverifikation, daher sollen sie in diesem Buch nicht weiter behandelt werden. An dieser Stelle seien jedoch eine kurze Charakterisierung und einige bibliographische Hinweise gegeben. Die bekannteste Methode dieser Art sind die *attribuierten Grammatiken* nach Knuth [64], die auf der Arbeit von Irons [54] aufbauen. Ausgangspunkt dieses Ansatzes ist die Zuordnung von sogenannten *Attributen* zu den syntaktischen Einheiten einer kontextfreien Grammatik. Die Werte dieser Attribute werden durch Angabe von *semantischen Funktionen* zu jeder Syntaxregel bestimmt.

Zur Einführung in das Konzept attribuierter Grammatiken eignen sich neben Knuth [64] auch Marcotty et al. [82], Lewis et al. [74], Räihä [103] und Pagan [99]. Beispiele für die Beschreibung der Semantik bekannter Programmiersprachen findet man u.a. bei Wilner [139] für SIMULA 67, bei Watt [131] für PASCAL und bei Uhl et al. [127] für Ada.

Attributierte Grammatiken haben ihre Hauptbedeutung als Hilfsmittel zur Beschreibung der semantischen Analyse und als Formalismus zur Beschreibung der

Standardisierung und Optimierung von Compilern (siehe z.B. Neel und Amirchahy [98], Babich und Jazayeri [7], Wilhelm [137], Waite und Goos [129] sowie Zima [141]).

Eine weitere Methode, die Syntax *und* die Semantik einer Programmiersprache mit *einem* Formalismus zu beschreiben, sind die *W-Grammatiken* oder *Zweistufiggrammatiken* nach van Wijngaarden [138]. Eine W-Grammatik besteht aus zwei Regelmengen, den Metaproduktionen und den Hyperregeln. Dabei sind die Hyperregeln kontextfrei mit parameterisierten, nichtterminalen Symbolen. Die Metaregeln sind ebenfalls kontextfrei und beschreiben die Wertebereiche dieser Parameter. Die Semantik wird nun implizit dadurch spezifiziert, daß genau alle zulässigen Kombinationen der Form (*Programm*, *Eingabe*, *Ausgabe*) syntaktisch mit einer W-Grammatik spezifiziert werden.

Van Wijngaarden et al. führen den Begriff der W-Grammatik ein und verwenden ihn zur Definition der Sprache ALGOL 68 (siehe [138] und auch die deutsche Übersetzung, Kerner [59]). Das Buch von Cleaveland und Uzgalis [26] sowie die Arbeit von Hesse [46] erklären anhand von zahlreichen Beispielen die Definition und die Anwendung von W-Grammatiken.

Ferner seien noch die von Ledgard entwickelten *Produktionssysteme* erwähnt ([72] und [73]). Ein Produktionssystem besteht wiederum aus zwei Grammatiken, wobei die erste die Syntax inklusive der Kontextabhängigkeiten spezifiziert und die zweite eine Abbildung definiert von der Menge der syntaktisch korrekten Programme in eine Zielsprache, deren Semantik als bekannt vorausgesetzt wird. Als Zielsprache kann sowohl direkt eine Maschinensprache gewählt werden als auch eine abstraktere Metasprache. Marcotty et al. spezifiziert in [82] eine Beispielsprache unter Verwendung von Produktionssystemen, wobei die Zielsprache auf der axiomatischen Semantik nach Hoare [48] basiert.

Schließlich sei noch auf die *Affixgrammatiken* von Koster [66] hingewiesen, die eng mit W-Grammatiken und Produktionssystemen verwandt sind. Crowe [28] und Watt [132] diskutieren die Anwendung von Affixgrammatiken im Rahmen des Compilerbaus.

Alle bisher angesprochenen Methoden eignen sich zur Formalisierung der Semantik *sequentieller* Programmiersprachen. Zur Behandlung von Parallelität (nebenläufige Programmausführung) entwickelt Milner [92] ein mathematisches Modell, CCS. Hoare stellt in [50] die nicht sequentielle Sprache CSP vor und formalisiert die Semantik nebenläufiger Programme. Weitere Ansätze zur Formalisierung von Parallelität gibt es im Rahmen der Theorie der Petri-Netze (siehe Brauer et al. [18] und Reisig [105]).

## Kapitel 2

# Verschiedene Methoden der formalen Semantikspezifikation

In diesem Kapitel wird eine kleine imperative Beispielsprache, WHILE, vorgestellt. Die einzige elementare Anweisungsform der Sprache WHILE ist die Zuweisung eines Wertes an eine Variable (*assignment*). Die komplexen Anweisungen lassen sich mittels Komposition (*sequencing*), Verzweigung (*conditional*) und `while`-Schleifen aufbauen.

Obwohl diese Sprache im Sinne der Berechenbarkeit arithmetischer Funktionen bereits universell ist, eignet sie sich nicht zum eleganten Programmieren, da wesentliche Strukturierungsmöglichkeiten wie komplexe Datenstrukturen, Blockstruktur und Prozedurkonzept fehlen. Der Sprachumfang ist jedoch ausreichend, um die Techniken der verschiedenen Semantikspezifikationsmethoden auf einer halbformalen Ebene kennenzulernen.

Nach der Spezifikation der formalen Syntax unserer Beispielsprache WHILE im nächsten Abschnitt soll die Semantik dieser Sprache zunächst informell in einem Stil, wie er in Handbüchern gängiger Programmiersprachen üblich ist, angegeben werden. Anschließend wird je eine formale Semantikspezifikation nach den Methoden operationell, denotationell und axiomatisch vorgestellt. Dabei sollen auch ihre Verwendungsmöglichkeiten und die Frage nach ihrer Äquivalenz zueinander diskutiert werden.

Die formale Semantik einer umfassenderen Sprache wird unter Verwendung der denotationellen Methode in Kapitel 4 ausführlich behandelt.

## 2.1 Die Beispielsprache WHILE

Die Sprache WHILE ist mit einer Ausnahme eine echte Teilsprache der Programmiersprache PASCAL (siehe Jensen und Wirth [55]). Die Ausnahme besteht darin, daß in der Sprache WHILE bereits syntaktisch zwischen *arithmetischen* und *booleschen Ausdrücken* unterschieden wird, während diese Unterscheidung bei PASCAL erst in der Semantik vorgenommen wird.

Zur Erzeugung syntaktisch korrekter WHILE-Programme dient eine kontextfreie Grammatik in BNF-ähnlicher Notation. Dabei werden alle aus dieser Grammatik erzeugbaren Phrasen in zehn syntaktische Bereiche unterteilt:

1. Die Menge *ZAHL* der Namen für ganze Zahlen, erzeugt aus dem Nichtterminal *Z*
2. die Menge *BOOL* der Namen für Wahrheitswerte, erzeugt aus dem Nichtterminal *W*
3. die Menge *KON* der Namen für Konstanten, erzeugt aus dem Nichtterminal *K*
4. die Menge *ID* (identifiers) der Variablen, erzeugt aus dem Nichtterminal *I*
5. die Menge *OP* der Symbole für zweistellige arithmetische Operationen, erzeugt aus dem Nichtterminal *OP*
6. die Menge *BOP* der Symbole für zweistellige boolesche Operationen, erzeugt aus dem Nichtterminal *BOP*
7. die Menge *TERM* der Terme (arithmetische Ausdrücke), erzeugt aus dem Nichtterminal *T*
8. die Menge *BT* der booleschen Ausdrücke, erzeugt aus dem Nichtterminal *B*
9. die Menge *COM* (commands) der Anweisungen, erzeugt aus dem Nichtterminal *C*
10. die Menge *PROG* der Programme, erzeugt aus dem Nichtterminal *P*.

Die Nichtterminalsymbole dienen mit und ohne Indizes auch als Metavariablen über den entsprechenden Bereichen.

### Definition 2.1 ( Syntax der Sprache WHILE )

#### (i) Elementare Einheiten

$$\begin{aligned}
 Z &::= \underline{0} \mid \underline{1} \mid \underline{2} \mid \cdots \mid \underline{-1} \mid \underline{-2} \mid \cdots \\
 W &::= \text{true} \mid \text{false} \\
 K &::= Z \mid W \\
 I &::= a \mid b \mid c \mid \cdots \mid x \mid a_1 \mid a_2 \mid \cdots \mid x_1 \mid x_2 \mid \cdots \\
 \underline{OP} &::= + \mid - \mid * \mid \div \mid \underline{\text{mod}} \\
 \underline{BOP} &::= = \mid < \mid > \mid \leq \mid \geq \mid \neq
 \end{aligned}$$

## (ii) Induktiv aufgebaute Einheiten

$$\begin{aligned}
 T &::= Z \mid I \mid T_1 \text{ OP } T_2 \mid \text{read} \\
 B &::= W \mid \text{not } B \mid T_1 \text{ BOP } T_2 \mid \text{read} \\
 C &::= \text{skip} \mid I := T \mid C_1; C_2 \mid \text{if } B \text{ then } C_1 \text{ else } C_2 \mid \text{while } B \text{ do } C \mid \\
 &\quad \text{output } T \mid \text{output } B \\
 P &::= C
 \end{aligned}$$

An dieser Stelle beobachtet man, daß die Sprache WHILE nicht eindeutig ist. Z.B. läßt sich eine Anweisung der Form

**while  $B$  do  $C_1; C_2$**

auf zwei verschiedene Weisen erzeugen:



Auch in der Sprache PASCAL existieren diese syntaktischen Mehrdeutigkeiten. Sie werden durch folgende Maßnahmen beseitigt: Zunächst werden Prioritäten gesetzt, die zu jedem Programm nur eine Zerlegung zulassen. So bindet z.B. die `while`-Schleife stärker als die Komposition (`;`), so daß etwa in obigem Beispiel nur die erste Zerlegung gültig ist. Bei den arithmetischen Ausdrücken vereinbart man die in der Mathematik üblichen Prioritäten, etwa `*` bindet stärker als `+`. Zusätzlich führt man Klammern ein, um eine andere als die durch die Prioritäten bestimmte Zerlegung zu erzwingen. So gibt es in PASCAL die gewohnten Klammern für Terme und die `begin - end` Klammern für Anweisungen. Wird etwa in obigem Beispiel die zweite Zerlegung gewünscht, so schreibt man:

```

 while  $B$  do
 begin
    $C_1;$ 
    $C_2$ 
 end
 
```

## Kapitel 2 Verschiedene Methoden der formalen Semantikspezifikation

Da in diesem Buch jedoch nicht die Syntaxanalyse im Vordergrund steht, wird im folgenden bei der Interpretation eines Programms stets angenommen, daß seine Struktur bekannt ist und es in seiner abstrakten Syntax vorliegt.

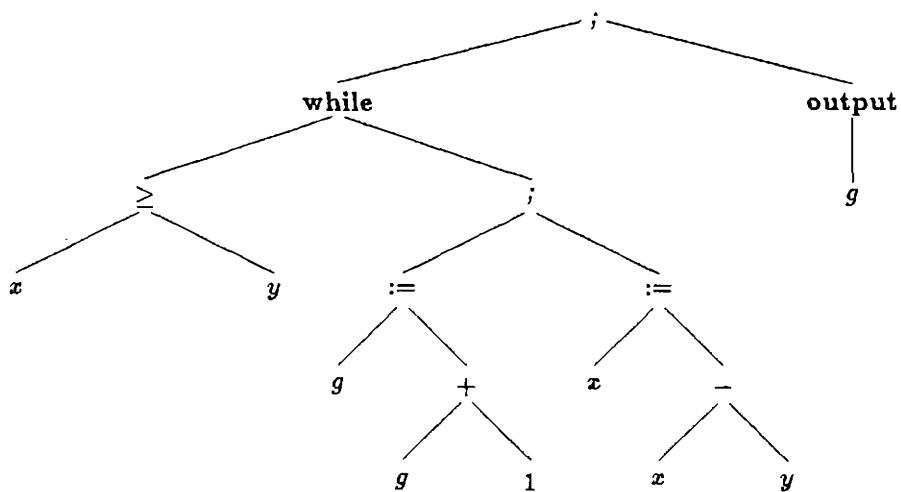
In Programmbeispielen wird die Struktur durch geeignete Einrückungen deutlich gemacht, während bei den Semantikspezifikationen ausschließlich davon Gebrauch gemacht wird, daß man auf die unmittelbaren Komponenten einer syntaktischen Einheit zugreifen kann. Die abstrakte Syntax eines Programms wird auch häufig durch den dazugehörigen *Syntaxbaum* dargestellt.

### Beispiel 2.2 (Ein Programm zur Berechnung der ganzzahligen Division)

In konkreter Syntax :

```
while x ≥ y do
    g := g + 1;
    x := x - y;
output g
```

In abstrakter Syntax :



Die Semantik einiger verbreiteter Programmiersprachen, wie z.B. FORTRAN IV, ist nur informell in natürlicher Sprache beschrieben. Eine solche Beschreibung von WHILE wird im nächsten Abschnitt angegeben.

## 2.2 Informelle Beschreibung der Semantik von WHILE

Der Interpretation von WHILE ist ein Speicherkonzept mit einer Ein- und einer Ausgabedatei zugrunde gelegt. Jede Variable bezeichnet die Adresse eines Speicherplatzes, und jeder Speicherplatz ist entweder frei oder er beinhaltet die Darstellung einer ganzen Zahl.

Die Definition der Bedeutung der syntaktischen Einheiten wird von folgenden Grundvorstellungen geleitet:

- Die Semantik eines *arithmetischen Ausdruckes* ist die ganze Zahl, die seinem Wert entspricht.
- Die Semantik eines *booleschen Ausdruckes* ist der Wahrheitswert, der seinem Wert entspricht.
- Die Semantik einer *Anweisung* ist die Speichertransformation, die durch ihre Ausführung bewirkt wird.
- Die Semantik eines *Programms* ist die Ein-/Ausgabefunktion, die durch die Semantik der dazugehörigen Anweisung bestimmt wird.

Es soll nun der Reihe nach die Bedeutung der syntaktischen Einheiten erklärt werden:

<i>Z</i>	Der Wert von <u>0</u> , <u>1</u> , <u>2</u> , ... <u>-1</u> , <u>-2</u> , ... ist die jeweils entsprechende ganze Zahl 0, 1, 2, ..., -1, -2, ... aus $\mathbb{Z}$ .
<i>W</i>	Der Wert von <b>true</b> bzw. <b>false</b> ist der dazugehörige Wahrheitswert <i>wahr</i> bzw. <i>falsch</i> .
<i>I</i>	Die Werte der Variablen sind dynamisch, d.h. sie lassen sich nur in Bezug auf den Zustand des Speichers erklären. So ist der momentane Wert einer Variablen <i>x</i> gleich dem momentanen Wert des Inhaltes des Speicherplatzes mit der symbolischen Adresse <i>x</i> .
<i>OP</i>	Die Bedeutungen der Zeichen <u>+</u> , <u>-</u> , <u>*</u> , <u>÷</u> und <u>mod</u> sind die entsprechenden mathematischen Operationen, wobei <u>mod</u> die Modulooperation bezeichnet, d.h. der Wert eines Ausdrückes der Form <i>a mod b</i> ist gleich dem Wert des Ausdrückes <i>a - ((a ÷ b) * b)</i> . Die Bedeutung von <u>÷</u> ist die ganzzahlige Division, wobei nicht gerundet wird. Führt die Anwendung einer Operation aus dem darstellbaren Zahlbereich hinaus, so ist das Ergebnis undefiniert.
<i>BOP</i>	Die Bedeutungen der Zeichen <u>=</u> , <u>&lt;</u> , <u>&gt;</u> , <u>≤</u> , <u>≥</u> und <u>≠</u> sind die entsprechenden Vergleichsoperationen, die von der Menge der darstellbaren ganzen Zahlen in die Menge der Wahrheitswerte erklärt sind.

**$T_1 \text{ } OP \text{ } T_2$**  Der Wert eines solchen Ausdruckes ist das Ergebnis der durch OP bezeichneten Funktion, angewendet auf die Werte von  $T_1$  und  $T_2$ .

**read** Der Wert von **read** ist die erste Konstante aus der Eingabedatei, falls diese nicht leer ist, andernfalls ist der Wert undefiniert. Die Berechnung von **read** hat die Nebenwirkung, daß die Eingabedatei um die erste Stelle verkürzt wird.

**not  $B$**  Der Wert eines Ausdrückes der Form **not  $B$**  ist die Negation des Wertes von  $B$ .

**$T_1 \text{ } BOP \text{ } T_2$**  Der Wert eines solchen Ausdruckes ist das Ergebnis der durch BOP bezeichneten Funktion, angewendet auf die Werte von  $T_1$  und  $T_2$ .

**skip** Die Ausführung von **skip** läßt den Speicher unverändert.

**$I := T$**  Die Ausführung einer solchen Wertzuweisung ändert den Speicher derart, daß der Wert von  $T$  in dem Speicherplatz mit der symbolischen Adresse  $I$  abgelegt wird.

**$C_1; C_2$**  Die zu  $C_1; C_2$  gehörende Speichertransformation ergibt sich aus der Hintereinanderausführung von  $C_1$  und  $C_2$ .

**if  $B$  then  $C_1$  else  $C_2$**  Die Ausführung einer solchen Verzweigung führt zur Ausführung von  $C_1$ , falls der Wert von  $B$  *wahr* ist, und zur Ausführung von  $C_2$ , falls der Wert von  $B$  *falsch* ist.

**while  $B$  do  $C$**  Die Speichertransformation, die durch die Ausführung einer solchen **while** - Schleife bewirkt wird, ist durch folgenden Algorithmus definiert:

Wenn die Berechnung des booleschen Ausdrückes  $B$  *wahr* ergibt, so wird erst die Anweisung  $C$  und anschließend noch einmal die gesamte Anweisung **while  $B$  do  $C$**  ausgeführt.

Ergibt die Berechnung von  $B$  *falsch*, so bleibt der Speicher unverändert und die Ausführung der gesamten Anweisung **while  $B$  do  $C$**  wird beendet.

**output  $T$**  Bei der Ausführung einer solchen Anweisung wird der Wert des arithmetischen Ausdrückes  $T$  berechnet und in die Ausgabedatei geschrieben.

**output  $B$**  Analog zur Anweisung **output  $T$**  wird hier der Wert des booleschen Ausdrückes  $B$  berechnet und in die Ausgabedatei geschrieben.

**$P$**  Die Semantik eines Programms  $P$ , das aus der Anweisung  $C$  besteht, ist die Ein-/Ausgabefunktion, die durch die von  $C$  bestimmte Speichertransformation definiert wird.

### 2.2.1 Diskussion der informellen Semantik von WHILE

Die oben angegebene Semantikbeschreibung legt einige Prinzipien der Programm-ausführung fest, läßt aber noch ein Spektrum unterschiedlicher Interpretationen zu. Folgende Aufzählung beschreibt die wichtigsten Bereiche, in denen Mehrdeutigkeiten auftreten:

#### Fehlerbehandlung

Bei der Programmausführung können folgende Fehler auftreten:

1. Bereichsüberschreitungen (Nur endlich viele Zahlen sind darstellbar.)
2. Division durch Null
3. Berechnung von `read` bei leerer Eingabedatei
4. Typkonflikte, z.B. `4 + read` bei einer Eingabedatei der Form `true .E`

Die Meldung von Fehlern und ihre Behandlung in der weiteren Programm-ausführung ist zwar in der Regel auch Gegenstand von informellen Semantikbeschreibungen, jedoch sind solche Beschreibungen wegen der Komplexität, die durch Fehlerfortpflanzungen entsteht, oft nicht vollständig.

#### Reihenfolge der Berechnung von Teilausdrücken

In der Regel werden die üblichen Prioritäten der arithmetischen Operationen bei der Berechnung von Ausdrücken berücksichtigt. Die Reihenfolge vom gleichrangigen Teilausdrücken kann jedoch für den Wert entscheidend sein. Betrachtet man etwa den Ausdruck

$$T_1 + T_2 - T_3 ,$$

so kann sich bei großem Wert von  $T_1$  oder  $T_2$  eine Bereichsüberschreitung bei der Berechnung der Summe ergeben. Bei hinreichend großem Wert von  $T_3$  tritt hingegen keine Bereichsüberschreitung auf, wenn erst die Differenz  $T_2 - T_3$  berechnet wird. Eine andere Situation, in der diese Reihenfolge von Bedeutung ist, entsteht bei der Berechnung von Teilausdrücken, bei deren Auswertung Nebeneffekte vorkommen. So ergibt die Berechnung von (`read - read`) bei einer Eingabedatei `5.2.E` je nach Reihenfolge der Berechnung der Unterausdrücke den Wert 3 oder -3.

Die meisten Benutzerhandbücher legen diese Reihenfolge nicht fest, so daß es compilerabhängig ist, ob gleichrangige Teilausdrücke in einer festen Reihenfolge ausgewertet werden oder ob diese Reihenfolge nach bestimmten Kriterien dynamisch bestimmt wird.

#### Behandlung komplexerer Sprachstrukturen

Die einfache, übersichtliche Struktur der Beispielsprache WHILE hat auch eine leicht durchschaubare Semantik zur Folge. Viele Unsauberheiten der informellen Semantikspezifikation ergeben sich bei der Behandlung höherer Sprachkonzepte, die hier noch nicht vorgestellt wurden. Typische Mehrdeutigkeiten entstehen durch Vereinbarung verschiedener Parameterübergabe-mechanismen und durch Nebeneffekte bei der Ausführung von Prozeduren.

### 2.3 Operationelle Semantik der Sprache WHILE

Die Spezifikation der operationellen Semantik einer Programmiersprache erfolgt durch die Definition einer abstrakten Maschine zur Simulation der Ausführung von Programmen. Zu einer solchen Definition gehören drei Angaben:

1. die Definition einer *Zustandsmenge*  $\mathcal{Z}$ , die aus geeigneten Informationsstrukturen aufgebaut ist,
2. eine Zuordnung, die zu jedem Programm  $P$  und jeder Eingabe  $E$  einen *Anfangszustand*  $z_{P,E} \in \mathcal{Z}$  bestimmt und
3. die Definition einer deterministischen *Zustandsüberführungsfunktion*

$$\Delta : \mathcal{Z} \longrightarrow \mathcal{Z}$$

zur Beschreibung des schrittweisen Effektes, den die Ausführung eines Programms auf der abstrakten Maschine auslöst.

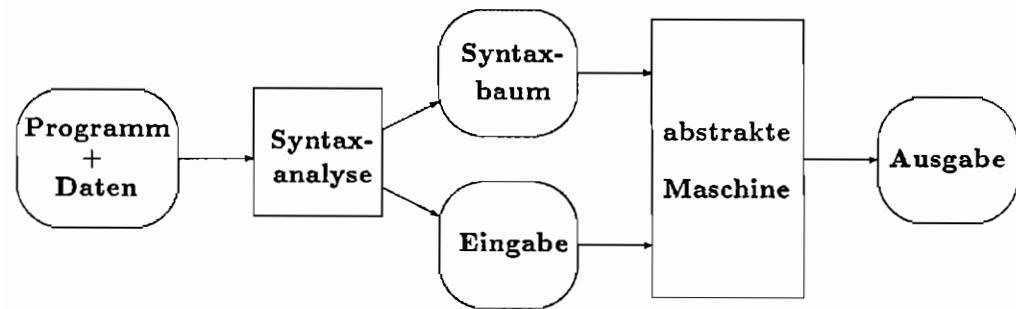
Aus diesen drei Angaben gewinnt man nun die operationelle Semantik einer Programm-Datenkombination  $(P, E)$  durch Iteration der Zustandsüberführungsfunktion  $\Delta$ , angewendet auf den Startzustand  $z_{P,E}$ , wie folgt:

Wenn die Funktion  $\Delta$  auf allen Folgezuständen definiert ist und jede einzelne Anwendung den Zustand verändert, so ist die Semantik der vorgelegten Programm-Datenkombination undefiniert, andernfalls ist sie definiert als die Ausgabekomponente desjenigen Zustandes  $z$ , für den gilt:

1.  $z = \Delta^n(z_{P,E})$  für ein  $n \in \mathbb{N}$ ,
2.  $\Delta(z) = z$  und
3.  $\Delta^k(z_{P,E}) \neq \Delta^{k+1}(z_{P,E})$  für alle  $k < n$ .

Dabei besagt die erste Bedingung, daß der Endzustand insbesondere ein Folgezustand des Anfangszustandes ist, die zweite Bedingung beschreibt die Terminierung der Programmausführung, und die letzte Bedingung garantiert die Eindeutigkeit der Semantikfunktion.

Folgendes Diagramm verdeutlicht in Form eines groben Schemas die operationelle Methode:



### 2.3.1 Die WSKEA-Maschine

Die WSKEA-Maschine ist eine abstrakte Maschine in obigem Sinne zur Ausführung von WHILE-Programmen.

Nun lauten der Reihe nach die drei benötigten Angaben:

- (i) **Die Zustandsmenge**  $\mathcal{Z}$  ist das kartesische Produkt aus einem Wertekeller  $W$ , einem Speicher  $S$ , einer endlichen Kontrolle  $K$ , einer Eingabedatei  $E$  und einer Ausgabedatei  $A$ . Ein Maschinenzustand wird daher durch ein Fünftupel  $< W \mid S \mid K \mid E \mid A >$  beschrieben.

Die einzelnen Komponenten haben folgende Struktur:

- $W$  ist ein Keller, in dem konstante Werte und Programmkomponenten (zur späteren Ausführung) abgelegt werden können (nicht zu verwechseln mit der Metavariable  $W$  für Wahrheitswerte).
- $S$  ist der Speicher, d.h. eine Zuordnung von Variablen zu ihren Werten. Er wird durch eine Abbildung vom Typ

$$ID \longrightarrow ZAHL \cup \{\underline{frei}\}$$

formalisiert.

- $K$  ist ein Keller (Kontrolle), in dem Programmkomponenten (zur Ausführung) und Symbole für spezielle Operationen (not, if, assign, while, output) abgelegt werden können (nicht zu verwechseln mit der Metavariable  $K$  für Konstanten).
- $E$  ist die Eingabedatei, d.h. eine Folge von Konstanten.
- $A$  ist die Ausgabedatei, ebenfalls eine Konstantenfolge.

**Bemerkung:** Eine äquivalente, implementierungsfreundlichere Maschinenstruktur ergäbe die Verwendung von Zeigern auf die Programmkomponenten, anstatt der Programmteile selbst. Die obige Struktur ist jedoch wegen der einfacheren Notation von Vorteil.

- (ii) **Der Anfangszustand**  $z_{P,E}$  zu einer Programm-Datenkombination  $(P,E)$  ist das Fünftupel

$$< \epsilon \mid S_0 \mid P.\epsilon \mid E \mid \epsilon >, \text{ wobei}$$

- $\epsilon$  den leeren Keller bzw. eine leere Datei bezeichnet,
- $S_0$  der leere Speicher ist, d.h.  $S_0(I) = \underline{frei}$  für alle  $I \in ID$  und
- $k.K$  den Keller bezeichnet, der aus aus dem Keller  $K$  durch Hinzufügen eines neuen Kellerelementes  $k$  entsteht.

**Bemerkung:** Im allgemeinen verwendet man die Punktschreibweise auch für Dateien und Listen. Seien etwa  $k, k_1, \dots, k_r \in KON$  ( $r \geq 0$ ), so wird die Eingabedatei  $(k, k_1, \dots, k_r)$  auch durch  $k.(k_1, \dots, k_r)$  bezeichnet oder die Ausgabedatei  $(k_1, \dots, k_r, k)$  durch  $(k_1, \dots, k_r).k$ .

- (iii) Die Zustandsüberführungsfunktion  $\Delta$  wird induktiv über den Aufbau derjenigen Programmkomponente, die auf der Spitze des Kontrollkellers  $K$  liegt, angegeben:

**Definition 2.3** Sei  $z \in \mathcal{Z}$ .

$$1. \underline{z = \langle W \mid S \mid T.K \mid E \mid A \rangle \text{ mit } T \in TERM}$$

- (a)  $\Delta \quad \langle W \mid S \mid \underline{n.K} \mid E \mid A \rangle := \langle \underline{n.W} \mid S \mid \underline{K} \mid E \mid A \rangle \quad \text{für alle } \underline{n} \in ZAHL$
- (b)  $\Delta \quad \langle W \mid S \mid \underline{x.K} \mid E \mid A \rangle := \langle S(x).W \mid S \mid \underline{K} \mid E \mid A \rangle \quad \text{für alle } x \in ID \text{ mit } S(x) \neq \underline{\text{frei}}$
- (c)  $\Delta \quad \langle W \mid S \mid \underline{T_1 OP T_2.K} \mid E \mid A \rangle := \langle W \mid S \mid \underline{T_1.T_2.OP.K} \mid E \mid A \rangle$
- (d)  $\Delta \quad \langle \underline{n_2.n_1.W} \mid S \mid \underline{+.K} \mid E \mid A \rangle := \langle \underline{n_1+n_2.W} \mid S \mid \underline{K} \mid E \mid A \rangle,$   
falls der Wert von  $n_1+n_2$  innerhalb des darstellbaren Zahlbereiches liegt
- (e) analog für alle anderen arithmetischen Operationen

$$(f) \quad \Delta \quad \langle W \mid S \mid \text{read}.K \mid \underline{n.E} \mid A \rangle := \langle \underline{n.W} \mid S \mid \underline{K} \mid \underline{E} \mid A \rangle \quad \text{für alle } \underline{n} \in ZAHL$$

$$2. \underline{z = \langle W \mid S \mid B.K \mid E \mid A \rangle \text{ mit } B \in BT}$$

- (a)  $\Delta \quad \langle W \mid S \mid \text{true}.K \mid E \mid A \rangle := \langle \text{true}.W \mid S \mid \underline{K} \mid E \mid A \rangle$
- (b)  $\Delta \quad \langle W \mid S \mid \text{false}.K \mid E \mid A \rangle := \langle \text{false}.W \mid S \mid \underline{K} \mid E \mid A \rangle$
- (c)  $\Delta \quad \langle W \mid S \mid \text{not}.B.K \mid E \mid A \rangle := \langle W \mid S \mid B.\underline{\text{not}}.K \mid E \mid A \rangle$

$$(d) \quad \Delta < b.W \mid S \mid \underline{not.K} \mid E \mid A > := \\ < \neg b.W \mid S \mid K \mid E \mid A > \quad \text{für alle } b \in \{\text{true , false}\},$$

$$\text{wobei } \neg b := \begin{cases} \text{false} & \text{falls } b = \text{true} \\ \text{true} & \text{falls } b = \text{false} \end{cases}$$

$$(e) \quad \Delta < W \mid S \mid T_1 \underline{BOP} T_2.K \mid E \mid A > := \\ < W \mid S \mid T_1.T_2.\underline{BOP}.K \mid E \mid A >$$

$$(f) \quad \Delta < \underline{n_2.n_1.W} \mid S \mid =.K \mid E \mid A > := \\ < \underline{n_1 = n_2.W} \mid S \mid K \mid E \mid A > , \\ \text{wobei } \underline{\text{wahr}} \text{ für true steht und } \underline{\text{falsch}} \text{ für false}$$

(g) analog für alle anderen booleschen Operationen

$$(h) \quad \Delta < W \mid S \mid \text{read .}K \mid b.E \mid A > := \\ < b.W \mid S \mid K \mid E \mid A > \quad \text{für alle } b \in \{\text{true , false}\}$$

3.  $z = < W \mid S \mid C.K \mid E \mid A >$  mit  $C \in COM$

$$(a) \quad \Delta < W \mid S \mid \text{skip .}K \mid E \mid A > := \\ < W \mid S \mid K \mid E \mid A >$$

$$(b) \quad \Delta < W \mid S \mid I := T.K \mid E \mid A > := \\ < I.W \mid S \mid T.\underline{\text{assign.}}K \mid E \mid A >$$

$$(c) \quad \Delta < \underline{n.I.W} \mid S \mid \underline{\text{assign.}}K \mid E \mid A > := \\ < W \mid S[n/I] \mid K \mid E \mid A > , \text{ wobei}$$

$$S[n/I](x) := \begin{cases} \frac{n}{S(x)} & \text{falls } x = I \\ S(x) & \text{falls } x \neq I \end{cases}$$

$$(d) \quad \Delta < W \mid S \mid C_1;C_2.K \mid E \mid A > := \\ < W \mid S \mid C_1.C_2.K \mid E \mid A >$$

$$(e) \quad \Delta < W \mid S \mid \text{if } B \text{ then } C_1 \text{ else } C_2.K \mid E \mid A > := \\ < W \mid S \mid B.\underline{\text{if.}}C_1.C_2.K \mid E \mid A >$$

$$(f) \quad \Delta < \text{true .}W \mid S \mid \underline{\text{if.}}C_1.C_2.K \mid E \mid A > := \\ < W \mid S \mid C_1.K \mid E \mid A >$$

$$(g) \quad \Delta < \mathbf{false} . W \mid S \mid \underline{\mathbf{if}.C_1.C_2.K} \mid E \mid A > := \\ < W \mid S \mid C_2.K \mid E \mid A >$$

$$(h) \quad \Delta < W \mid S \mid \mathbf{while} B \mathbf{do} C.K \mid E \mid A > := \\ < C.B.W \mid S \mid B.\underline{\mathbf{while}.K} \mid E \mid A >$$

$$(i) \quad \Delta < \mathbf{true} . C.B.W \mid S \mid \underline{\mathbf{while}.K} \mid E \mid A > := \\ < W \mid S \mid C;\mathbf{while} B \mathbf{do} C.K \mid E \mid A >$$

$$(j) \quad \Delta < \mathbf{false} . C.B.W \mid S \mid \underline{\mathbf{while}.K} \mid E \mid A > := \\ < W \mid S \mid K \mid E \mid A >$$

$$(k) \quad \Delta < W \mid S \mid \mathbf{output} T.K \mid E \mid A > := \\ < W \mid S \mid T.\underline{\mathbf{output}.K} \mid E \mid A >$$

$$(l) \quad \Delta < \underline{n}.W \mid S \mid \underline{\mathbf{output}.K} \mid E \mid A > := \\ < W \mid S \mid K \mid E \mid A.\underline{n} >$$

(m) analog für **output B**

$$4. \quad z = < W \mid S \mid \varepsilon \mid E \mid A >$$

$$\Delta < W \mid S \mid \varepsilon \mid E \mid A > := < W \mid S \mid \varepsilon \mid E \mid A >$$

■

Man prüft leicht nach, daß die Zustandsüberführungsfunktion  $\Delta$  deterministisch, aber nicht vollständig definiert ist, und daß eine Programmausführung beendet ist, wenn der Kontrollkeller leer ist oder kein Folgezustand existiert.

Die formale Semantik eines Programms wird meist nicht als Funktion auf den *Namen* für die Konstanten, sondern direkt auf den mathematischen Wertebereichen definiert. Daher bezeichnet  $\overline{KON}$  die entsprechende Menge der Konstanten, d.h.:

$$\mathbb{Z} := \{\dots, -2, -1, 0, 1, 2, \dots\},$$

$$\overline{BOOL} := \{wahr, falsch\} \text{ und}$$

$$\overline{KON} := \mathbb{Z} \cup \overline{BOOL}.$$

Analog bezeichnet  $\bar{k}$  den Wert von  $k$  für alle  $k \in KON$ , und das Tupel  $(\bar{k}_1, \dots, \bar{k}_n)$  bezeichnet das Wertetupel  $(\bar{k}_1, \dots, \bar{k}_n)$  für alle  $k_\nu \in KON$  mit  $1 \leq \nu \leq n$ ,  $n \in IN$ .

Die operationelle Semantik von WHILE läßt sich nun präzise als partielle Funktion definieren.

**Definition 2.4 (Die operationelle Semantik der Sprache WHILE)**

$$\mathcal{O} : PROG \longrightarrow [\overline{KON^*} \longrightarrow \overline{KON^*} \cup \{\underline{Fehler}\}]$$

$$\mathcal{O}(P)(\overline{E}) := \begin{cases} \overline{A} & \text{falls } \Delta^n(z_{P,E}) = \Delta^{n+1}(z_{P,E}) \text{ für ein } n \in IN \text{ und} \\ & \Pi_5(\Delta^n(z_{P,E})) = A \\ \underline{Fehler} & \text{falls für ein } n \in IN \Delta^n(z_{P,E}) \text{ nicht definiert ist} \\ \underline{undefiniert} & \text{sonst} \end{cases}$$

wobei für alle  $i \in IN$ ,  $\Pi_i$  die Projektion auf die  $i$ -te Komponente bezeichnet.

■

Zur Illustration dieser Definition soll die Ausführung eines kurzen WHILE-Programms auf der WSKEA-Maschine nachvollzogen werden.

**Beispiel 2.5 (Ein Programm zur Berechnung der Division mit Rest)**

$$C \left\{ \begin{array}{l} C_1 \left\{ \begin{array}{l} x := \text{read;} \\ y := \text{read;} \\ g := 0; \end{array} \right. \\ \text{while } x \geq y \text{ do} \\ \quad C_2 \left\{ \begin{array}{l} g := g + 1; \\ C_4 \left\{ \begin{array}{l} x := x - y; \end{array} \right. \end{array} \right. \\ \quad C_3 \left\{ \begin{array}{l} \text{output } g; \\ \text{output } x \end{array} \right. \end{array} \right. \right.$$

Die Markierung auf der linken Seite einer geschweiften Klammer soll die rechts daneben stehenden Anweisungen unter einem Namen zusammenfassen.

Zur Berechnung von 7 dividiert durch 5 bildet man den Anfangszustand  $z_{C,7.5.e}$  und wendet  $\Delta$  so lange an, bis die Kontrollkomponente leer ist:

$$\begin{aligned} z_{C,7.5.e} &= < \epsilon \mid S_0 \mid C.e \mid \underline{7.5.e} \mid \epsilon > \\ \xrightarrow{3d} &< \epsilon \mid S_0 \mid C_1.C_2; C_3.e \mid \underline{7.5.e} \mid \epsilon > \\ \xrightarrow{3d} &< \epsilon \mid S_0 \mid x := \text{read}.y := \text{read}; g := 0.C_2; C_3.e \mid \underline{7.5.e} \mid \epsilon > \\ \xrightarrow{3b} &< x.e \mid S_0 \mid \text{read}.\underline{\text{assign.y := read}}; g := 0.C_2; C_3.e \mid \underline{7.5.e} \mid \epsilon > \\ \xrightarrow{1f} &< \underline{7.x.e} \mid S_0 \mid \underline{\text{assign.y := read}}; g := 0.C_2; C_3.e \mid \underline{5.e} \mid \epsilon > \\ \xrightarrow{3c} &< \epsilon \mid S_0[\underline{7}/x] \mid y := \text{read}; g := 0.C_2; C_3.e \mid \underline{5.e} \mid \epsilon > \end{aligned}$$

$$\begin{aligned}
& \xrightarrow{3d} <\epsilon \mid S_0[\underline{7}/x] \mid y := \mathbf{read}.g := \underline{0}.C_2; C_3.\epsilon \mid \underline{5}.\epsilon \mid \epsilon> \\
& \xrightarrow{3b} <y.\epsilon \mid S_0[\underline{7}/x] \mid \mathbf{read}.\underline{assign}.g := \underline{0}.C_2; C_3.\epsilon \mid \underline{5}.\epsilon \mid \epsilon> \\
& \xrightarrow{1f} <\underline{5}.y.\epsilon \mid S_0[\underline{7}/x] \mid \underline{assign}.g := \underline{0}.C_2; C_3.\epsilon \mid \epsilon \mid \epsilon> \\
& \xrightarrow{3c} <\epsilon \mid S_0[\underline{7}/x][\underline{5}/y] \mid g := \underline{0}.C_2; C_3.\epsilon \mid \epsilon \mid \epsilon> \\
& \xrightarrow{3b} <g.\epsilon \mid S_0[\underline{7}/x][\underline{5}/y] \mid \underline{0}.assign.C_2; C_3.\epsilon \mid \epsilon \mid \epsilon> \\
& \xrightarrow{1a} <\underline{0}.g.\epsilon \mid S_0[\underline{7}/x][\underline{5}/y] \mid assign.C_2; C_3.\epsilon \mid \epsilon \mid \epsilon> \\
& \xrightarrow{3c} <\epsilon \mid S_0[\underline{7}/x][\underline{5}/y][\underline{0}/g] \mid C_2; C_3.\epsilon \mid \epsilon \mid \epsilon> \\
& \xrightarrow{3d} <\epsilon \mid S_0[\underline{7}/x][\underline{5}/y][\underline{0}/g] \mid \mathbf{while} \ x \geq y \mathbf{do} \ C_4.C_3.\epsilon \mid \epsilon \mid \epsilon> \\
& \xrightarrow{3h} <C_4.x \geq y.\epsilon \mid S_0[\underline{7}/x][\underline{5}/y][\underline{0}/g] \mid x \geq y.\underline{while}.C_3.\epsilon \mid \epsilon \mid \epsilon> \\
& \xrightarrow{2e} <C_4.x \geq y.\epsilon \mid S_0[\underline{7}/x][\underline{5}/y][\underline{0}/g] \mid x.y. \geq .\underline{while}.C_3.\epsilon \mid \epsilon \mid \epsilon> \\
& \xrightarrow{1b} <\underline{7}.C_4.x \geq y.\epsilon \mid S_0[\underline{7}/x][\underline{5}/y][\underline{0}/g] \mid y. \geq .\underline{while}.C_3.\epsilon \mid \epsilon \mid \epsilon> \\
& \xrightarrow{1b} <\underline{5}.\underline{7}.C_4.x \geq y.\epsilon \mid S_0[\underline{7}/x][\underline{5}/y][\underline{0}/g] \mid \geq .\underline{while}.C_3.\epsilon \mid \epsilon \mid \epsilon> \\
& \xrightarrow{2g} <\mathbf{true}.C_4.x \geq y.\epsilon \mid S_0[\underline{7}/x][\underline{5}/y][\underline{0}/g] \mid \underline{while}.C_3.\epsilon \mid \epsilon \mid \epsilon> \\
& \xrightarrow{3i} <\epsilon \mid S_0[\underline{7}/x][\underline{5}/y][\underline{0}/g] \mid C_4; C_2.C_3.\epsilon \mid \epsilon \mid \epsilon> \\
& \xrightarrow{3d} <\epsilon \mid S_0[\underline{7}/x][\underline{5}/y][\underline{0}/g] \mid C_4.C_2.C_3.\epsilon \mid \epsilon \mid \epsilon> \\
& \xrightarrow{3d} <\epsilon \mid S_0[\underline{7}/x][\underline{5}/y][\underline{0}/g] \mid g := g + \underline{1}.x := x - y.C_2.C_3.\epsilon \mid \epsilon \mid \epsilon> \\
& \xrightarrow{3b} <g.\epsilon \mid S_0[\underline{7}/x][\underline{5}/y][\underline{0}/g] \mid g + \underline{1}.assign.x := x - y.C_2.C_3.\epsilon \mid \epsilon \mid \epsilon> \\
& \xrightarrow{1c} <g.\epsilon \mid S_0[\underline{7}/x][\underline{5}/y][\underline{0}/g] \mid g.\underline{1}. + .assign.x := x - y.C_2.C_3.\epsilon \mid \epsilon \mid \epsilon> \\
& \xrightarrow{1b} <\underline{0}.g.\epsilon \mid S_0[\underline{7}/x][\underline{5}/y][\underline{0}/g] \mid \underline{1}. + .assign.x := x - y.C_2.C_3.\epsilon \mid \epsilon \mid \epsilon> \\
& \xrightarrow{1a} <\underline{1}.\underline{0}.g.\epsilon \mid S_0[\underline{7}/x][\underline{5}/y][\underline{0}/g] \mid +.assign.x := x - y.C_2.C_3.\epsilon \mid \epsilon \mid \epsilon> \\
& \xrightarrow{1d} <\underline{1}.g.\epsilon \mid S_0[\underline{7}/x][\underline{5}/y][\underline{0}/g] \mid assign.x := x - y.C_2.C_3.\epsilon \mid \epsilon \mid \epsilon>
\end{aligned}$$

$\xrightarrow{3c}$   $< \epsilon \mid S_0[7/x][5/y][1/g] \mid x := x - y.C_2.C_3.\epsilon \mid \epsilon \mid \epsilon >$   
 $\xrightarrow{3b}$   $< x.\epsilon \mid S_0[7/x][5/y][1/g] \mid x - y.\underline{\text{assign}}.C_2.C_3.\epsilon \mid \epsilon \mid \epsilon >$   
 $\xrightarrow{1c}$   $< x.\epsilon \mid S_0[7/x][5/y][1/g] \mid x.y. - .\underline{\text{assign}}.C_2.C_3.\epsilon \mid \epsilon \mid \epsilon >$   
 $\xrightarrow{1b}$   $< 7.x.\epsilon \mid S_0[7/x][5/y][1/g] \mid y. - .\underline{\text{assign}}.C_2.C_3.\epsilon \mid \epsilon \mid \epsilon >$   
 $\xrightarrow{1b}$   $< 5.7.x.\epsilon \mid S_0[7/x][5/y][1/g] \mid -.\underline{\text{assign}}.C_2.C_3.\epsilon \mid \epsilon \mid \epsilon >$   
 $\xrightarrow{1e}$   $< 2.x.\epsilon \mid S_0[7/x][5/y][1/g] \mid \underline{\text{assign}}.C_2.C_3.\epsilon \mid \epsilon \mid \epsilon >$   
 $\xrightarrow{3c}$   $< \epsilon \mid S_0[5/y][1/g][2/x] \mid C_2.C_3.\epsilon \mid \epsilon \mid \epsilon >$   
 $\xrightarrow{3h}$   $< C_4.x \geq y.\epsilon \mid S_0[5/y][1/g][2/x] \mid x \geq y.\underline{\text{while}}.C_3.\epsilon \mid \epsilon \mid \epsilon >$   
 $\xrightarrow{2e}$   $< C_4.x \geq y.\epsilon \mid S_0[5/y][1/g][2/x] \mid x.y. \geq .\underline{\text{while}}.C_3.\epsilon \mid \epsilon \mid \epsilon >$   
 $\xrightarrow{1b}$   $< 2.C_4.x \geq y.\epsilon \mid S_0[5/y][1/g][2/x] \mid y. \geq .\underline{\text{while}}.C_3.\epsilon \mid \epsilon \mid \epsilon >$   
 $\xrightarrow{1b}$   $< 5.2.C_4.x \geq y.\epsilon \mid S_0[5/y][1/g][2/x] \mid \geq .\underline{\text{while}}.C_3.\epsilon \mid \epsilon \mid \epsilon >$   
 $\xrightarrow{2g}$   $< \text{false}.C_4.x \geq y.\epsilon \mid S_0[5/y][1/g][2/x] \mid \underline{\text{while}}.C_3.\epsilon \mid \epsilon \mid \epsilon >$   
 $\xrightarrow{3j}$   $< \epsilon \mid S_0[\underline{5}/y][1/g][2/x] \mid C_3.\epsilon \mid \epsilon \mid \epsilon >$   
 $\xrightarrow{3d}$   $< \epsilon \mid S_0[\underline{5}/y][1/g][2/x] \mid \text{output } g.\text{output } x.\epsilon \mid \epsilon \mid \epsilon >$   
 $\xrightarrow{3k}$   $< \epsilon \mid S_0[\underline{5}/y][1/g][2/x] \mid g.\underline{\text{output}}.\text{output } x.\epsilon \mid \epsilon \mid \epsilon >$   
 $\xrightarrow{1b}$   $< 1.\epsilon \mid S_0[\underline{5}/y][1/g][2/x] \mid \underline{\text{output}}.\text{output } x.\epsilon \mid \epsilon \mid \epsilon >$   
 $\xrightarrow{3l}$   $< \epsilon \mid S_0[\underline{5}/y][1/g][2/x] \mid \text{output } x.\epsilon \mid \epsilon \mid \epsilon.1 >$   
 $\xrightarrow{3k}$   $< \epsilon \mid S_0[\underline{5}/y][1/g][2/x] \mid x.\underline{\text{output}}.\epsilon \mid \epsilon \mid \epsilon.1 >$   
 $\xrightarrow{1b}$   $< 2.\epsilon \mid S_0[\underline{5}/y][1/g][2/x] \mid \underline{\text{output}}.\epsilon \mid \epsilon \mid \epsilon.1 >$   
 $\xrightarrow{3l}$   $< \epsilon \mid S_0[\underline{5}/y][1/g][2/x] \mid \epsilon \mid \epsilon \mid \epsilon.1.2 >$   
 $\xrightarrow{4}$   $< \epsilon \mid S_0[\underline{5}/y][1/g][2/x] \mid \epsilon \mid \epsilon \mid \epsilon.1.2 >$

Also ist die Semantik dieses Programms unter obiger Eingabe

$$\mathcal{O}(C)((7, 5)) = (1, 2) \quad .$$

### 2.3.2 Diskussion der operationellen Semantik von WHILE

Mit der Definition der Semantikfunktion  $\mathcal{O}$  ist nun die Bedeutung von WHILE-Programmen eindeutig als das Ergebnis ihrer Ausführung auf der WSKEA-Maschine festgelegt.

Dies soll noch einmal für die in Abschnitt 2.2.1 aufgeführten Problemstellungen vergegenwärtigt werden:

#### Fehlerbehandlung

Die WSKEA-Maschine sieht die einfachste Form der Fehlerbehandlung vor: Führt eine Programmausführung auf eine Konfiguration, für die die Zustandsübergitungsfunktion  $\Delta$  nicht definiert ist, d.h. für die kein Folgezustand existiert, so ist die Bedeutung dieses Programms gleich „Fehler“. Natürlich wird man für eine praxisorientierte Programmiersprache differenzierte Fehlermeldungen erzeugen wollen. Entsprechende Modifikationen sind auch leicht vorzunehmen, wenn man aus der partiellen Funktion  $\Delta$  eine deterministische Funktion

$$\Delta' : \mathcal{Z} \longrightarrow \mathcal{Z} \cup \{\underline{\text{Fehler}}_1, \dots, \underline{\text{Fehler}}_n\}$$

entwickelt und zu jedem Fehlerotyp einen Kommentar schreibt:

#### Beispiel 2.6 (Fehlerbehandlung)

$\Delta' < \underline{n_2.n_1}.W \mid S \mid +.K \mid E \mid A > := \underline{\text{Fehler}}_1$ , falls der Wert von  $n_1 + n_2$  außerhalb des darstellbaren Zahlbereiches liegt

$\Delta' < \underline{n_2.n_1}.W \mid S \mid /.K \mid E \mid A > := \underline{\text{Fehler}}_2$ , falls  $n_2 = 0$

$\Delta' < W \mid S \mid \text{read}.K \mid \varepsilon \mid A > := \underline{\text{Fehler}}_3$

$\Delta' < b.\underline{n}.W \mid S \mid +.K \mid E \mid A > := \underline{\text{Fehler}}_4$ , falls  $b \in \{\text{true}, \text{false}\}$

und  $n \in \mathbb{Z}$

#### Kommentare:

Fehler<sub>1</sub> entsteht bei Auftreten eines „Overflow“,

Fehler<sub>2</sub> entsteht bei Division durch Null,

Fehler<sub>3</sub> entsteht bei Lesezugriff auf eine leere Eingabedatei und

Fehler<sub>4</sub> entsteht bei Auftreten eines Typkonfliktes.

Es kommt uns aber für die Beispielsprache WHILE nicht darauf an, eine praktikable Sprache zu entwerfen, sondern aufzuzeigen, wie eine *eindeutige* Semantikspezifikation im operationellen Stil entwickelt werden kann. Daher ist die einfachste Form der Fehlerbehandlung hier ausreichend.

#### Reihenfolge der Berechnung von Teilausdrücken

Die WSKEA-Maschine schreibt eine strikte Auswertungsreihenfolge von links nach rechts vor (siehe Definition 2.3.1c und 2e). Auch hier ist es einfach, Modifikationen für bestimmte Situationen anzubringen. Wichtig ist jedoch auch hier, daß genau eine Reihenfolge *eindeutig* festgelegt ist.

Wie sich am Beispiel 2.5 gut erkennen läßt, führt die Verwendung der Einzelschrittfunktion zu einem mühsamen Nachvollziehen der Zustandsänderungen der WSKEA-Maschine. Damit lassen sich sicher noch nicht elegant Programmeigenschaften beweisen, sondern die Funktion  $\Delta$  dient eher dazu, Testläufe zu simulieren. Man kann diese Situation dadurch verbessern, daß man aus der operationellen Semantikbeschreibung durch Abstraktion von geeigneten Folgen von Zwischenschritten eine *Reduktionssemantik* gewinnt.

### 2.3.3 Die Reduktionssemantik der Sprache WHILE

Um das globale Verhalten der WSKEA-Maschine besser in den Griff zu bekommen, definiert man eine *Reduktionsrelation* „ $\Rightarrow$ “ über dem kartesischen Produkt aus syntaktischen Einheiten und dem Zustandskonzept.

Eine informelle Beschreibung dieser Relation lautet: Eine Phrase (syntaktische Einheit)  $P$  und ein Tripel  $(S, E, A)$  stehen in der Reduktionsrelation „ $\Rightarrow$ “ mit einer Phrase  $P'$  und einem Tripel  $(S', E', A')$ , wenn die Ausführung von  $P$  bzgl. des Speichers  $S$ , der Eingabe  $E$  und der Ausgabe  $A$  den „Wert“  $P'$  ergibt und den Speicher in  $S'$ , die Eingabe in  $E'$  und die Ausgabe in  $A'$  transformiert. Eine formale Axiomatisierung der Relation „ $\Rightarrow$ “ läßt sich induktiv über den Aufbau der WHILE-Phrasen angeben:

#### Definition 2.7 (Reduktionsrelation $\Rightarrow$ )

##### 1. TERM

- (a)  $(x, (S, E, A)) \Rightarrow (S(x), (S, E, A))$  , falls  
 $S(x) \neq \underline{frei}$
- (b)  $(T_1 \text{ OP } T_2, (S, E, A)) \Rightarrow (T'_1 \text{ OP } T_2, (S, E', A))$  , falls  
 $(T_1, (S, E, A)) \Rightarrow (T'_1, (S, E', A))$
- (c)  $(\underline{n} \text{ OP } T, (S, E, A)) \Rightarrow (\underline{n} \text{ OP } T', (S, E', A))$  , falls  
 $(T, (S, E, A)) \Rightarrow (T', (S, E', A))$
- (d)  $(\underline{n}_1 + \underline{n}_2, (S, E, A)) \Rightarrow (\underline{n}_1 + \underline{n}_2, (S, E, A))$  , falls

der Wert von  $n_1 + n_2$  innerhalb des darstellbaren Zahlbereiches liegt

(e) analog für alle anderen arithmetischen Operationen

(f)  $(\text{read}, (S, \underline{n}.E, A)) \Rightarrow (\underline{n}, (S, E, A))$

### 2. BT

(a)  $(\text{not } B, (S, E, A)) \Rightarrow (\text{not } B', (S, E', A))$ , falls  
 $(B, (S, E, A)) \Rightarrow (B', (S, E', A))$

(b)  $(\text{not } b, (S, E, A)) \Rightarrow (\neg b, (S, E, A))$  für  
 $b \in \{\text{true}, \text{false}\}$

(c)  $(T_1 \text{ BOP } T_2, (S, E, A))$  analog zu 1b - 1e

(d)  $(\text{read}, (S, b.E, A)) \Rightarrow (b, (S, E, A))$

### 3. COM

(a)  $(\text{skip} ; C, (S, E, A)) \Rightarrow (C, (S, E, A))$

(b)  $(I := T, (S, E, A)) \Rightarrow (\text{skip}, (S[\underline{n}/I], E', A))$ , f  
 $(T, (S, E, A)) \xrightarrow{*} (\underline{n}, (S, E', A))$ , wobei  
 $\xrightarrow{*}$  die reflexive, transitive Hülle von  $\Rightarrow$  bezeichnet.

(c)  $(C_1; C_2, (S, E, A)) \Rightarrow (C'_1; C_2, (S', E', A'))$ , falls  
 $(C_1, (S, E, A)) \Rightarrow (C'_1, (S', E', A'))$

(d)  $(\text{if } B \text{ then } C_1 \text{ else } C_2, (S, E, A)) \Rightarrow (C_1, (S, E', A))$ , falls  
 $(B, (S, E, A)) \xrightarrow{*} (\text{true}, (S, E', A))$

(e)  $(\text{if } B \text{ then } C_1 \text{ else } C_2, (S, E, A)) \Rightarrow (C_2, (S, E', A))$ , falls  
 $(B, (S, E, A)) \xrightarrow{*} (\text{false}, (S, E', A))$

(f)  $(\text{while } B \text{ do } C, (S, E, A)) \Rightarrow (C; \text{while } B \text{ do } C, (S, E', A))$ , falls  
 $(B, (S, E, A)) \xrightarrow{*} (\text{true}, (S, E', A))$

(g)  $(\text{while } B \text{ do } C, (S, E, A)) \Rightarrow (\text{skip}, (S, E', A))$ , falls  
 $(B, (S, E, A)) \xrightarrow{*} (\text{false}, (S, E', A))$

(h)  $(\text{output } T, (S, E, A)) \Rightarrow (\text{skip}, (S, E', A.\underline{n}))$ , falls  
 $(T, (S, E, A)) \xrightarrow{*} (\underline{n}, (S, E', A))$

(i)  $(\text{output } B, (S, E, A)) \Rightarrow (\text{skip}, (S, E', A.b))$ , falls  
 $(B, (S, E, A)) \xrightarrow{*} (b, (S, E', A))$

■

Man überzeugt sich leicht, daß die Relation „ $\Rightarrow$ “ deterministisch ist. Damit läßt sich die operationelle Semantikfunktion auch auf dieser Reduktionsrelation begründen:

#### Definition 2.8 (Die Reduktionssemantik der Sprache WHILE)

$eval : PROG \rightarrow [KON^* \rightarrow KON^* \cup \{\text{Fehler}\}]$

$$eval(P)(\bar{E}) := \begin{cases} \bar{A} & \text{falls } (C, (S_0, E, \epsilon)) \xrightarrow{*} (\text{skip}, (S, E', A)) \text{ mit } S \text{ und } E' \text{ beliebig} \\ \underline{\text{Fehler}} & \text{falls } (C, (S_0, E, \epsilon)) \xrightarrow{*} (C', (S, E', A)) \text{ mit } C' \neq \text{skip}, S, E' \text{ und } A \text{ beliebig, und } (C', (S, E', A)) \text{ läßt sich nicht weiter mit } " \Rightarrow " \text{ reduzieren} \\ \text{undefined} & \text{sonst} \end{cases}$$

■

Der Vorteil dieser Reduktionssemantik gegenüber der operationellen Semantik besteht darin, daß beim Argumentieren über ein Programmverhalten nicht mehr jeder einzelne Schritt der WSKEA-Maschine betrachtet werden muß, da jeweils eine Reduktion bereits eine sinnvolle Zusammenfassung mehrerer Maschinenschritte simuliert. Für eine Implementierung hingegen ist sicherlich die operationelle Methode günstiger, da hier die Festlegung einer schrittweisen Programmausführung erforderlich ist. Daher ist es wünschenswert, zu einer operationellen Semantikdefinition auch eine Reduktionssemantik anzugeben und die Äquivalenz beider Definitionen nachzuweisen. Dann kann je nach Erfordernis die eine oder die andere Spezifikation verwendet werden.

#### 2.3.4 Äquivalenz von operationeller und Reduktionssemantik

Die Standardbeweismethode bei induktiven Definitionen ist die strukturelle Induktion. Im allgemeinen bedeutet dies, daß man eine Eigenschaft zunächst für die elementaren syntaktischen Einheiten nachweist und dann diese Eigenschaft für die induktiv aufgebauten Einheiten herleitet, indem man von den unmittelbaren Komponenten einer syntaktischen Einheit diese Eigenschaft voraussetzt.

Diese Methode läßt sich jedoch für den gewünschten Äquivalenzbeweis nicht anwenden, da die Definitionen zur Behandlung einer Schleife `while B do C` nicht nur auf den Bedeutungen von `B` und `C` aufbauen, sondern rekursiv auch auf die Bedeutung der ganzen `while`-Schleife zurückgreifen. Man kann aber die Äquivalenz der Semantikspezifikationen  $\mathcal{O}$  und  $eval$  durch Induktion über die Ausführungsänge zeigen.

Zunächst soll in zwei Lemmata gezeigt werden, daß die WSKEA-Maschine genau dann mit einem Ergebnis terminiert, wenn dieses Ergebnis auch vermöge der Reduktionsrelation hergeleitet werden kann.

**Lemma 2.9 (Die WSKEA-Maschine befolgt die Relation  $\Rightarrow$ )**

Sei  $m \in \mathbb{N}$  und  $\langle W \mid S \mid K \mid E \mid A \rangle$  ein beliebiger Zustand.

**1. Berechnung arithmetischer Terme:**

Wenn  $(T, (S, E, A)) \xrightarrow{m} (\underline{n}, (S', E', A'))$ , dann gilt  $S = S'$ ,  $A = A'$ , und es gibt ein  $k \in \mathbb{N}$  mit

$$\Delta^k \langle W \mid S \mid T.K \mid E \mid A \rangle = \langle \underline{n}.W \mid S \mid K \mid E' \mid A \rangle.$$

**2. Berechnung boolescher Terme:**

Wenn  $(B, (S, E, A)) \xrightarrow{m} (b, (S', E', A'))$ , dann gilt  $S = S'$ ,  $A = A'$ , und es gibt ein  $k \in \mathbb{N}$  mit

$$\Delta^k \langle W \mid S \mid B.K \mid E \mid A \rangle = \langle b.W \mid S \mid K \mid E' \mid A \rangle.$$

**3. Ausführung von Anweisungen:**

Wenn  $(C, (S, E, A)) \xrightarrow{m} (\text{skip}, (S', E', A'))$ , dann gibt es ein  $k \in \mathbb{N}$  mit

$$\Delta^k \langle W \mid S \mid C.K \mid E \mid A \rangle = \langle W \mid S' \mid K \mid E' \mid A' \rangle.$$

**Beweis:**

**1. Arithmetische Terme:**

**Induktionsanfang ( $m = 0$ )**

Es gilt  $T = \underline{n}$ ,  $S' = S$ ,  $E' = E$  und  $A' = A$ , also folgt die Behauptung mit  $k = 0$ .

**Induktionsschritt**

Annahme: Die Behauptung gilt bereits für alle  $m' < m > 0$ .

**Fallunterscheidung:**

$T = x$

Es gilt  $(x, (S, E, A)) \Rightarrow (S(x), (S, E, A))$  nach Definition 2.7.1a und ebenso

$$\Delta \langle W \mid S \mid x.K \mid E \mid A \rangle = \langle S(x).W \mid S \mid K \mid E \mid A \rangle \\ \text{nach Definition 2.3.1b.}$$

Also gilt die Behauptung mit  $k = 1$ .

$T = T_1 \text{ OP } T_2$

Aus  $(T, (S, E, A)) \xrightarrow{m} (\underline{n}, (S', E', A'))$  folgt nach  
Definition 2.7.1b - 1e

- (a)  $(T_1, (S, E, A)) \xrightarrow{m_1} (\underline{n}_1, (S, E'', A))$  und  
 (b)  $(T_2, (S, E'', A)) \xrightarrow{m_2} (\underline{n}_2, (S, E', A))$

mit  $m_1 + m_2 + 1 = m$ ,  $S = S'$  und  $A = A'$ .

Die Induktionsannahme besagt nun, daß es natürliche Zahlen  $k_1$  und  $k_2$  gibt mit

- (a)  $\Delta^{k_1} < W \mid S \mid T_1.K \mid E \mid A > =$   
 $< \underline{n}_1.W \mid S \mid K \mid E'' \mid A >$  und  
 (b)  $\Delta^{k_2} < W \mid S \mid T_2.K \mid E'' \mid A > =$   
 $< \underline{n}_2.W \mid S \mid K \mid E' \mid A > .$

Nun gilt:

$$\Delta < W \mid S \mid T_1 \text{OP} T_2.K \mid E \mid A > = \\ < W \mid S \mid T_1.T_2.K \mid E \mid A > \quad \text{nach Definition 2.3.1c}$$

$$\xrightarrow{\Delta^{k_1}} < \underline{n}_1.W \mid S \mid T_2.\text{OP}.K \mid E'' \mid A > \quad \text{nach Induktionsannahme} \\ \xrightarrow{\Delta^{k_2}} < \underline{n}_2.\underline{n}_1.W \mid S \mid \text{OP}.K \mid E' \mid A > \quad \text{nach Induktionsannahme} \\ \xrightarrow{\Delta} < \underline{n}.W \mid S \mid K \mid E' \mid A > \quad \text{nach Definition 2.3.1d bzw. 1e}.$$

Also gilt die Behauptung mit  $k = k_1 + k_2 + 1$ .

### $T = \text{read}$

Aus der Definition von  $\implies$  (2.7.1f) folgt:

$E = \underline{n}.E'$ ,  $S = S'$  und  $A = A'$ , wobei

$$(\text{read}, (S, \underline{n}.E', A)) \implies (\underline{n}, (S, E', A)) ,$$

und aus der Definition von  $\Delta$  (2.3.1f) folgt:

$$\Delta < W \mid S \mid \text{read}.K \mid \underline{n}.E' \mid A > = \\ < \underline{n}.W \mid S \mid K \mid E' \mid A > .$$

Also gilt die Behauptung mit  $k = 1$ .

2. Boolesche Terme: analog zu 1.

3. Anweisungen:

### Induktionsanfang ( $m = 0$ )

In diesem Fall gilt  $C = \text{skip}$ ,  $S' = S$ ,  $E' = E$  und  $A' = A$  und nach

Definition 2.3.3a gilt auch

$$\Delta < W \mid S \mid \text{skip}.K \mid E \mid A > = < W \mid S \mid K \mid E \mid A > .$$

Also folgt die Behauptung mit  $k = 1$ .

### Induktionsschritt

Annahme: Die Behauptung gilt bereits für alle  $m' < m > 0$  und

$$(C, (S, E, A)) \xrightarrow{m} (\text{skip}, (S', E', A')).$$

Fallunterscheidung:

$C = \text{skip}$

Dieser Fall ist nicht möglich, da  $(\text{skip}, (S, E, A))$  nicht mit  $\Rightarrow$  weiter reduziert werden kann.

$C = I := T$

Aus Definition 2.7.3b folgt  $m = 1$ ,  $S' = S[n/I]$ ,  $A = A'$  und

$$(T, (S, E, A)) \xrightarrow{*} (\underline{n}, (S, E', A)) .$$

Aus 1. folgt: Es gibt ein  $k \in \mathbb{N}$  mit

$$\Delta^k < W \mid S \mid T.K \mid E \mid A > = < \underline{n}.W \mid S \mid K \mid E' \mid A > .$$

Nun gilt:

$$\begin{aligned} \Delta &< W \mid S \mid I := T.K \mid E \mid A > = \\ &< I.W \mid S \mid T.\underline{\text{assign}}.K \mid E \mid A > \quad \text{nach Definition 2.3.3b} \end{aligned}$$

$$\begin{aligned} &\xrightarrow{\Delta^k} < \underline{n}.I.W \mid S \mid \underline{\text{assign}}.K \mid E' \mid A > \\ &\xrightarrow{\Delta} < W \mid S[n/I] \mid K \mid E' \mid A > \quad \text{nach Definition 2.3.3c.} \end{aligned}$$

Also gilt die Behauptung mit  $k + 2$ .

$$\frac{C = C_1; C_2 \mid \text{if } B \text{ then } C_1 \text{ else } C_2 \mid \text{output } T \mid \text{output } B}{\text{Als Übung}}$$

$C = \text{while } B \text{ do } C_1$

Aus Definition 2.7.3f und 3g folgt

$$(B, (S, E, A)) \xrightarrow{*} (b, (S, E'', A)) \quad \text{mit } b \in \{\text{true}, \text{false}\}$$

und

$$(C, (S, E, A)) \xrightarrow{} \begin{cases} (C_1; C, (S, E'', A)) & \text{falls } b = \text{true} \\ (\text{skip}, (S, E'', A)) & \text{falls } b = \text{false} \end{cases} .$$

Aus obiger Annahme folgt nun, daß die rechte Seite in  $m - 1$  Schritten auf

$(\text{skip}, (S', E', A'))$

reduziert werden kann.

Ferner gilt:

$$\begin{aligned}
 \Delta < W \mid S \mid C.K \mid E \mid A > \\
 = & < C_1.B.W \mid S \mid B.\underline{\text{while}}.K \mid E \mid A > \text{ nach Definition 2.3.3h} \\
 \xrightarrow{\Delta^{k_1}} & < b.C_1.B.W \mid S \mid \underline{\text{while}}.K \mid E'' \mid A > \text{ nach Teil 2.} \\
 \xrightarrow{\Delta} & \begin{cases} < W \mid S \mid C_1; C.K \mid E'' \mid A > & \text{falls } b = \text{true} \\ < W \mid S \mid K \mid E'' \mid A > & \text{falls } b = \text{false} \end{cases} \\
 & \text{nach Definition 2.3.3i und 3j} \\
 \xrightarrow{\Delta^{k_2}} & < W \mid S' \mid K \mid E' \mid A' > \\
 & \text{nach Induktionsannahme für } m - 1.
 \end{aligned}$$

Also gilt die Behauptung mit  $k = k_1 + k_2 + 2$ .

Q.E.D.

Lemma 2.9 ist die eine Hälfte der gewünschten Äquivalenz. Für die andere Hälfte benötigt man noch einen Begriff, der die Tatsache formalisiert, daß die Ausführung eines Ausdrückes auf der WSKEA-Maschine nicht auf Kellerinhalte zugreift, die bei Beginn der Ausführung in dem Kontrollspeicher  $K$  vorhanden sind.

#### Definition 2.10 (Perfekte Ausführungsfolge)

Sei  $D$  eine syntaktische Phrase der Sprache WHILE.

Eine Ausführungsfolge

$$< W \mid S \mid D.K \mid E \mid A > \xrightarrow{\Delta^*} < W' \mid S' \mid K \mid E' \mid A' >$$

heißt perfekt, wenn der Kontrollkeller nach jedem Zwischenschritt eine echte Erweiterung von  $K$  ist, d.h. erst nach dem letzten Schritt ist der Kontrollkeller gleich  $K$ . ■

#### Lemma 2.11 (Die Relation $\Rightarrow$ simuliert die WSKEA-Maschine )

Sei  $k \in IN$  und  $< W \mid S \mid K \mid E \mid A >$  ein beliebiger Zustand.

##### 1. Berechnung arithmetischer Terme:

Wenn die Ausführungsfolge

$$\langle W \mid S \mid T.K \mid E \mid A \rangle \xrightarrow{\Delta^k} \langle W' \mid S' \mid K \mid E' \mid A' \rangle$$

perfekt ist, dann gilt:  $W' = \underline{n}.W$  für ein  $n \in \mathbb{Z}$ ,  $S' = S$ ,  $A' = A$  und

$$(T, (S, E, A)) \xrightarrow{*} (\underline{n}, (S, E', A)) .$$

## 2. Berechnung boolescher Terme:

Wenn die Ausführungsfolge

$$\langle W \mid S \mid B.K \mid E \mid A \rangle \xrightarrow{\Delta^k} \langle W' \mid S' \mid K \mid E' \mid A' \rangle$$

perfekt ist, dann gilt:  $W' = b.W$  für ein  $b \in \{\text{true, false}\}$ ,  $S' = S$ ,  $A' = A$  und

$$(B, (S, E, A)) \xrightarrow{*} (b, (S, E', A)) .$$

## 3. Ausführung von Anweisungen:

Wenn die Ausführungsfolge

$$\langle W \mid S \mid C.K \mid E \mid A \rangle \xrightarrow{\Delta^k} \langle W' \mid S' \mid K \mid E' \mid A' \rangle$$

perfekt ist, dann gilt:  $W' = W$  und

$$(C, (S, E, A)) \xrightarrow{*} (\text{skip}, (S', E', A')) .$$

## Beweis:

### 1. Arithmetische Terme:

**Induktionsanfang ( $k = 0$ )**

In diesem Fall ist die Prämisse falsch, also die Aussage wahr.

#### Induktionsschritt

Annahme: Die Behauptung gilt bereits für alle  $m' < m > 0$  und die Ausführung

$$\langle W \mid S \mid T.K \mid E \mid A \rangle \xrightarrow{\Delta^k} \langle W' \mid S' \mid K \mid E' \mid A' \rangle$$

ist perfekt.

#### Fallunterscheidung:

$T = x$

Es gilt

$$\langle W \mid S \mid x.K \mid E \mid A \rangle \xrightarrow{\Delta} \langle S(x).W \mid S \mid K \mid E \mid A \rangle$$

nach Definition 2.3.1b

und ebenso  $(x, (S, E, A)) \xrightarrow{\text{Def. 2.7.1a}} (S(x), (S, E, A))$  nach Definition 2.7.1a.

$$\underline{T = T_1 \text{ OP } T_2}$$

Es gilt:

$$\begin{aligned} \Delta < W \mid S \mid T_1 \text{ OP } T_2.K \mid E \mid A > = \\ < W \mid S \mid T_1.T_2.\text{OP}.K \mid E \mid A > \quad \text{nach Definition 2.3.1c} \end{aligned}$$

$$\begin{aligned} \xrightarrow{\Delta^{k_1}} & < \underline{n_1}.W \mid S \mid T_2.\text{OP}.K \mid E'' \mid A > \\ & \text{perfekt nach Definition 2.3.1a-1f} \\ \xrightarrow{\Delta^{k_2}} & < \underline{n_2}.\underline{n_1}.W \mid S \mid \text{OP}.K \mid E' \mid A > \\ & \text{perfekt nach Definition 2.3.1a-1f} \\ \xrightarrow{\Delta} & < \underline{n}.W \mid S \mid K \mid E' \mid A > \quad \text{nach Definition 2.3.1d bzw. 1e.} \end{aligned}$$

Aus der Induktionsannahme folgt:

- (a)  $(T_1, (S, E, A)) \xrightarrow{*} (\underline{n_1}, (S, E'', A))$  und
- (b)  $(T_2, (S, E'', A)) \xrightarrow{*} (\underline{n_2}, (S, E', A))$ .

Nun folgt aus Definition 2.7.1b-1e auch

$$(T, (S, E, A)) \xrightarrow{*} (\underline{n}, (S, E', A)).$$

$$\underline{T = \text{read}}$$

Aus der Definition (2.3.1f) folgt:

$$E = \underline{n}.E', S = S' \text{ und } A = A', \text{ wobei}$$

$$\begin{aligned} \Delta & < W \mid S \mid \text{read}.K \mid \underline{n}.E' \mid A > = \\ & < \underline{n}.W \mid S \mid K \mid E' \mid A > , \end{aligned}$$

und aus der Definition 2.7.1f folgt

$$(\text{read}, (S, \underline{n}.E', A)) \xrightarrow{} (\underline{n}, (S, E', A)).$$

Also gilt die Behauptung.

2. Boolesche Terme: analog zu 1.

3. Anweisungen:

**Induktionsanfang ( $k = 0$ )**

In diesem Fall ist die Prämisse falsch, also die Aussage wahr.

**Induktionsschritt**

Annahme: Die Behauptung gilt bereits für alle  $k' < k > 0$  und  
 $\langle W \mid S \mid C.K \mid E \mid A \rangle \xrightarrow{\Delta^k} \langle W' \mid S' \mid K \mid E' \mid A' \rangle$  ist  
eine perfekte Ausführung.

**Fallunterscheidung:**

$C = \text{skip}$

In diesem Fall muß  $k = 1$  sein, und es gilt:  $W' = W$ ,  $S' = S$ ,  
 $E' = E$  und  $A' = A$ .

Wegen der Reflexivität von  $\xrightarrow{*}$  gilt auch

$$(\text{skip}, (S, E, A)) \xrightarrow{*} (\text{skip}, (S, E, A)).$$

$C = I := T$

Aus Definition 2.3.3b und 3c folgt  $\langle W \mid S \mid C.K \mid E \mid A \rangle$

$$\begin{aligned} &\xrightarrow{\Delta} \langle I.W \mid S \mid T.\underline{\text{assign}}.K \mid E \mid A \rangle \\ &\xrightarrow{\Delta^{k-2}} \langle W'' \mid S'' \mid \underline{\text{assign}}.K \mid E'' \mid A \rangle \\ &\xrightarrow{\Delta} \langle W' \mid S' \mid K \mid E' \mid A \rangle. \end{aligned}$$

Da diese Ausführungsfolge perfekt ist, folgt aus 1:

$$W'' = \underline{n}.W, S'' = S \text{ und } (T, (S, E, A)) \xrightarrow{*} (n, (S, E'', A)) .$$

Aus Definition 2.3.3c folgt nun  $W' = W$ ,  $S' = S[\underline{n}/I]$  und  $E'' = E$ .

Mit Definition 2.7.3b folgt  $(C, (S, E, A)) \xrightarrow{*} (\text{skip}, (S', E', A)) .$

$C = C_1; C_2 \mid \text{if } B \text{ then } C_1 \text{ else } C_2 \mid \text{output } T \mid \text{output } B$   
Als Übung

$C = \text{while } B \text{ do } C_1$

Aus Definition 2.3.3h - 3j und 2 folgt:

$$\begin{aligned} &\langle W \mid S \mid C.K \mid E \mid A \rangle \\ &\xrightarrow{\Delta} \langle C_1.B.W \mid S \mid B.\underline{\text{while}}.K \mid E \mid A \rangle \\ &\xrightarrow{\Delta^{k_1}} \langle W'' \mid S'' \mid \underline{\text{while}}.K \mid E'' \mid A \rangle \\ &= \langle b.C_1.B.W \mid S \mid \underline{\text{while}}.K \mid E'' \mid A \rangle \text{ mit } b \in \{\text{true}, \text{false}\} \\ &\xrightarrow{\Delta} \left\{ \begin{array}{ll} \langle W \mid S \mid C_1; C.K \mid E'' \mid A \rangle & \text{falls } b = \text{true} \\ \langle W \mid S \mid K \mid E'' \mid A \rangle & \text{falls } b = \text{false} \end{array} \right. \end{aligned}$$

$$\xrightarrow{\Delta^{k_2}} \langle W' \mid S' \mid K \mid E' \mid A' \rangle .$$

Da  $k_2 < k$  gilt, kann man die Induktionsannahme anwenden und erhält:

$$(*) \begin{cases} (C_1; C, (S, E'', A)) \xrightarrow{*} (\text{skip}, (S', E', A')) \text{ bzw.} \\ (\text{skip}, (S, E'', A)) \xrightarrow{*} (\text{skip}, (S', E', A')) . \end{cases}$$

Ferner folgt

$$(B, (S, E, A)) \xrightarrow{*} (b, (S, E'', A)) \text{ aus Teil 2 dieses Lemmas.}$$

Damit folgt aus Definition 2.7.3f bzw. 3g

$$(C, (S, E, A)) \xrightarrow{*} \begin{cases} (C_1; C, (S, E'', A)) \text{ falls } b = \text{true} \\ (\text{skip}, (S, E'', A)) \text{ falls } b = \text{false} \end{cases}$$

Zusammen mit (\*) ergibt sich das gewünschte Resultat.

Q.E.D.

Mit diesen beiden Lemmata ist es nun möglich, die Äquivalenz der operationellen und der Reduktionssemantik zu zeigen.

### Satz 2.12 (Äquivalenz von operationeller und Reduktionssemantik)

$$\mathcal{O} = eval$$

**Beweis:**

Es genügt, folgende Äquivalenzen zu zeigen:

Für alle Speicher  $S$  und  $S'$ , Eingaben  $E$  und  $E'$  und Ausgaben  $A$  gilt

1.  $\langle \varepsilon \mid S \mid C.\varepsilon \mid E \mid \varepsilon \rangle \xrightarrow{\Delta^*} \langle \varepsilon \mid S' \mid \varepsilon \mid E' \mid A \rangle \text{ gdw}$   
 $(C, (S, E, \varepsilon)) \xrightarrow{*} (\text{skip}, (S', E', A))$
2.  $\langle \varepsilon \mid S \mid C.\varepsilon \mid E \mid \varepsilon \rangle \xrightarrow{\Delta^*} \langle W \mid S' \mid K \mid E' \mid A \rangle \text{ mit}$   
 $\Delta \langle W \mid S' \mid K \mid E' \mid A \rangle \text{ ist nicht definiert}$   
 $\text{gdw}$   
 $(C, (S, E, \varepsilon)) \xrightarrow{*} (C', (S', E', A)) \text{ mit}$   
 $(C', ((S', E', A))) \text{ lässt sich nicht weiter mit } \Rightarrow \text{ reduzieren.}$

zu 1. „ $\Rightarrow$ “: Angenommen, daß

$$\langle \varepsilon \mid S \mid C.\varepsilon \mid E \mid \varepsilon \rangle \xrightarrow{\Delta^*} \langle \varepsilon \mid S' \mid \varepsilon \mid E' \mid A \rangle$$

gilt, dann muß diese Ausführung perfekt sein. Also ist Lemma 2.11.3 anwendbar.

„ $\Leftarrow$ “: Eine Anwendung des Lemmas 2.9.3 liefert die gewünschte Eigenschaft.

zu 2. Man prüft leicht nach, daß zu jeder Konfiguration der WSKEA-Maschine, für die  $\Delta$  nicht definiert ist, ein entsprechendes Tupel existiert, das nicht mit  $\Rightarrow$  weiter reduziert werden kann.

Q.E.D.

Die Definition der Semantik von WHILE mittels der Funktion *eval* ermöglicht es, Programmeigenschaften eleganter zu beweisen, da bereits von einigen Implementierungsdetails, wie Werte- und Kontrollkeller, abstrahiert wurde.

Dies soll an dem Programm aus Beispiel 2.5 demonstriert werden:

### Beispiel 2.13 (Korrektheitsbeweis zu 2.5)

Sei  $C$  das WHILE-Programm aus Beispiel 2.5 zur Berechnung der ganzzahligen Division mit Rest:

$$C \left\{ \begin{array}{l} C_1 \left\{ \begin{array}{l} x := \text{read}; \\ y := \text{read}; \\ g := 0; \end{array} \right. \\ C_2 \left\{ \begin{array}{l} \text{while } x \geq y \text{ do} \\ C_4 \left\{ \begin{array}{l} g := g + 1; \\ x := x - y; \end{array} \right. \end{array} \right. \\ C_3 \left\{ \begin{array}{l} \text{output } g; \\ \text{output } x \end{array} \right. \end{array} \right.$$

**Voraussetzung:** Seien  $n, m \in \mathbb{N}$  und  $m > 0$ .

**Behauptung:**  $\text{eval}(C)(n, m) = (q, r)$  mit

$$(*) \quad n = q \cdot m + r \quad \text{und} \quad r < m.$$

**Beweis:** Nach Definition 2.8 von eval genügt es zu zeigen, daß

$$(C, (S_0, \underline{n.m.\varepsilon}, \varepsilon)) \xrightarrow{*} (\text{skip}, (S, \varepsilon, \varepsilon.q.r)) \quad \text{mit } (*)$$

gilt.

Aus den Regeln 2.7.3c, 1f, 3a und 3b folgt:

$$(C, (S_0, \underline{n.m.\varepsilon}, \varepsilon)) \xrightarrow{*} (C_2; C_3, (S_0 [n/x] [m/y] [0/g], \varepsilon, \varepsilon)) .$$

Sei  $z_1 = (S_1, \varepsilon, \varepsilon)$  mit  $S_1 = S_0[n/x][m/y][0/g]$  und sei induktiv  $z_{\nu+1}$  der Folgezustand von  $z_\nu$  nach einmaligem Durchlaufen der while - Schleife  $C_2$ , d.h. unter der Annahme, daß

$$(x \geq y, z_\nu) \xrightarrow{*} (\text{true}, z_\nu)$$

gilt, folgt

$$(C_4, z_\nu) \xrightarrow{*} (\text{skip}, z_{\nu+1}) = (\text{skip}, (S_{\nu+1}, \varepsilon, \varepsilon)) .$$

Man zeigt induktiv:

$$(**) \quad n = \overline{S_\nu(g)} \cdot m + \overline{S_\nu(x)} .$$

**Induktionsanfang ( $\nu = 1$ )**

$$\text{Es gilt: } n = 0 \cdot m + n = \overline{S_1(g)} \cdot m + \overline{S_1(x)}.$$

**Induktionsschritt**

$$\text{Es gelte: } (x \geq y, z_\nu) \xrightarrow{*} (\text{true}, z_\nu).$$

Dann folgt aus Definition 2.7.3c, 3b und 1a - 1e:

$$(C_4, z_\nu) \xrightarrow{*} (\text{skip}, (S_{\nu+1}, \varepsilon, \varepsilon)),$$

$$\text{wobei } S_{\nu+1} = S_\nu [\overline{S_\nu(g)} + 1/g] [\overline{S_\nu(x)} - \overline{S_\nu(y)}/x].$$

$$\text{Damit gilt: } \overline{S_{\nu+1}(g)} \cdot m + \overline{S_{\nu+1}(x)}$$

$$= (\overline{S_\nu(g)} + 1) \cdot m + \overline{S_\nu(x)} - \overline{S_\nu(y)}$$

$$= \overline{S_\nu(g)} \cdot m + m + \overline{S_\nu(x)} - m$$

$$= n \text{ nach Induktionsvoraussetzung.}$$

Unter der Annahme, daß  $m > 0$  gilt, gibt es nur endlich viele Schleifendurchläufe, da  $S_\nu(y)$  immer konstant gleich  $m$  bleibt, aber  $S_\nu(x)$  bei jedem Schleifendurchlauf um  $m$  kleiner wird. Also gibt es ein  $k \in \mathbb{N}$  mit:

1.  $(C, (S_0, \underline{n.m.\varepsilon}, \varepsilon)) \xrightarrow{*} (C_2; C_3, (S_k, \varepsilon, \varepsilon)) ,$
2.  $(x \geq y, (S_k, \varepsilon, \varepsilon)) \xrightarrow{*} (\text{false}, (S_k, \varepsilon, \varepsilon)) \text{ und}$
3.  $(**) \text{ mit } \nu = k.$

Nun folgt aus Definition 2.7.3c, 3g, 3h und 1a:

$$(C, (S_0, \underline{n.m.\varepsilon}, \varepsilon)) \xrightarrow{*} (\text{skip}, (S_k, \varepsilon, S_k(g).S_k(x).\varepsilon)) .$$

Mit  $q = S_k(g)$  und  $r = S_k(x)$  gilt die erste Hälfte von (\*).

Da  $\overline{S_k(x)} < \overline{S_k(y)}$  ist, gilt auch  $r < m$ , die zweite Hälfte von (\*).

Q.E.D.

## 2.4 Denotationelle Semantik der Sprache WHILE

In diesem Abschnitt soll auf einer halbformalen Ebene die Semantik der Sprache WHILE im denotationellen Stil entwickelt werden. Wie schon in der Einleitung beschrieben, ist es das Ziel der denotationellen Methode, jedem Programm *direkt* die dazugehörige Ein-/Ausgabefunktion zuzuordnen, ohne die schrittweise Zustandsänderung einer Maschine zu berücksichtigen. Dazu wird zunächst zu jedem syntaktischen Bereich der Sprache ein geeigneter semantischer Bereich definiert, und anschließend werden die semantischen Funktionen angegeben, die jedem syntaktischen Objekt das dazugehörige semantische Objekt zuordnen. In der Regel werden diese semantischen Bereiche und Funktionen durch Gleichungen spezifiziert. Wie man aus solchen Gleichungen eindeutige Lösungen gewinnt, wird im folgenden Kapitel genau behandelt. Hier sollen die Ideen der denotationellen Methode vermittelt werden, ohne dabei auf den mathematischen Hintergrund einzugehen.

Als erstes soll ein mathematischer Bereich zur Formalisierung des Zustandskonzeptes definiert werden:

### Definition 2.14 (Zustandsbereich)

1.  $ZUSTAND = SPEICHER \times EINGABE \times AUSGABE$
2.  $SPEICHER - ID \longrightarrow (\mathbb{Z} \cup \{\underline{frei}\})$
3.  $EINGABE = \overline{KON}^*$
4.  $AUSGABE = \overline{KON}^*$

### Erläuterung:

1. besagt, daß der semantische Bereich der Zustände aus allen Tripeln der Form  $(s, e, a)$  besteht, wobei  $s$  ein Element aus dem Bereich SPEICHER ist,  $e$  ein Element aus dem Bereich EINGABE und  $a$  ein Element aus dem Bereich AUSGABE.
2. besagt, daß der semantische Bereich SPEICHER aus allen Funktionen von dem Bereich ID in den Bereich  $(\mathbb{Z} \cup \{\underline{frei}\})$  besteht.
3. und 4. schließlich bedeuten, daß die semantischen Bereiche EINGABE und AUSGABE aus den endlichen Folgen (inklusive der leeren Folge) von Konstanten (Elementen aus  $\overline{KON}$ ) bestehen.

Der Unterschied zwischen dem mathematischen Bereich ZUSTAND und dem Zustandskonzept der operationellen und Reduktionssemantik besteht ausschließlich in der Verwendung des mathematischen Wertebereiches der Konstanten anstelle von

*ihren Namen. Während eine abstrakte Maschine nur auf den Namen für Konstanten arbeiten kann, darf der Zielbereich der semantischen Funktionen die Menge der Konstanten selbst enthalten.*

■

Für die Sprache WHILE sollen nun vier Semantikfunktionen definiert werden, die folgende Typen haben:

$$\begin{aligned} T &: TERM \longrightarrow \{\text{Bedeutung der Terme}\}, \\ B &: BT \longrightarrow \{\text{Bedeutung der booleschen Terme}\}, \\ C &: COM \longrightarrow \{\text{Bedeutung der Anweisungen}\} \text{ und} \\ P &: PROG \longrightarrow \{\text{Bedeutung der Programme}\}, \end{aligned}$$

wobei die in Mengenklammern genannten Bereiche noch anzugeben sind.

Für die Terme und booleschen Terme würde man zunächst die Menge der ganzen Zahlen bzw. der Wahrheitswerte als semantische Bereiche vermuten. Die folgenden vier Überlegungen führen jedoch zu komplexeren Strukturen:

1. Die Verwendung von Variablen in den Ausdrücken lässt eine direkte Zuordnung von Termen zu Konstanten nicht zu. Den Funktionen  $T$  und  $B$  muß also noch ein Parameter aus dem Bereich  $ZUSTAND$  hinzugegeben werden.
2. Die so entstehenden Typen

$$\begin{aligned} T &: TERM \times ZUSTAND \longrightarrow \mathbb{Z} \text{ bzw.} \\ B &: BT \times ZUSTAND \longrightarrow \overline{BOOL} \end{aligned}$$

sind für eine Verwendung im Rahmen der denotationellen Methode nicht geeignet, da man nicht mehr von der Bedeutung eines Termes sprechen kann, sondern nur noch von der Bedeutung einer Kombination aus einem Term und einem Zustand. Daher wendet man die nach Curry benannte Isomorphie

$$\begin{aligned} \underline{\text{curry}} : (A \times B \longrightarrow C) &\longrightarrow (A \longrightarrow (B \longrightarrow C)) \text{ mit} \\ \underline{\text{curry}}(f)(a)(b) &:= f(a, b) \end{aligned}$$

an, um eine Semantikfunktion für Terme zu erhalten. So entstehen die folgenden Typen:

$$\begin{aligned} T &: TERM \longrightarrow (ZUSTAND \longrightarrow \mathbb{Z}) \text{ und} \\ B &: BT \longrightarrow (ZUSTAND \longrightarrow \overline{BOOL}). \end{aligned}$$

3. Da die Berechnung von Ausdrücken auch den Zustand verändern kann (in der Sprache WHILE nur die Eingabekomponente), müssen die Zielbereiche der Funktionen  $T$  und  $B$  auch noch eine zweite Komponente aus dem Bereich

*ZUSTAND* zur Beschreibung des Folgezustandes haben. Damit erhält man folgende Typen:

$$\begin{aligned} T &: TERM \longrightarrow (ZUSTAND \longrightarrow (\mathbb{Z} \times ZUSTAND)) \text{ und} \\ B &: BT \longrightarrow (ZUSTAND \longrightarrow (\overline{BOOL} \times ZUSTAND)). \end{aligned}$$

4. Um die Behandlung von Fehlern zu ermöglichen, soll ein zusätzliches Element Fehler in den Wertebereich hinzugefügt werden. Damit ergibt sich der semantische Bereich

$$(ZUSTAND \longrightarrow ((\mathbb{Z} \times ZUSTAND) \cup \{\underline{Fehler}\}))$$

zur Formalisierung der Bedeutung der Terme und der semantische Bereich

$$(ZUSTAND \longrightarrow ((\overline{BOOL} \times ZUSTAND) \cup \{\underline{Fehler}\}))$$

zur Formalisierung der Bedeutung der booleschen Ausdrücke.

Die Überlegungen zur Definition der semantischen Bereiche für die Bedeutung der Anweisungen und Programme ergeben sich aus den Angaben der informellen Semantik. So wird der semantische Bereich der Bedeutung der Anweisungen die Menge der Funktionen von *ZUSTAND* in *ZUSTAND*  $\cup \{\underline{Fehler}\}$  sein, und der semantische Bereich der Bedeutung von Programmen wird, wie bereits aus der operationellen Semantik bekannt, der Bereich der Ein-/Ausgabefunktionen von *KON\** in *KON\**  $\cup \{\underline{Fehler}\}$  sein.

### Definition 2.15 (Semantikfunktionen)

Die Semantikfunktionen der Sprache WHILE haben folgende Typen:

$$\begin{aligned} T &: TERM \longrightarrow (ZUSTAND \longrightarrow ((\mathbb{Z} \times ZUSTAND) \cup \{\underline{Fehler}\})), \\ B &: BT \longrightarrow (ZUSTAND \longrightarrow ((\overline{BOOL} \times ZUSTAND) \cup \{\underline{Fehler}\})), \\ C &: COM \longrightarrow (ZUSTAND \longrightarrow (ZUSTAND \cup \{\underline{Fehler}\})) \text{ und} \\ P &: PROG \longrightarrow (\overline{KON^*} \longrightarrow (\overline{KON^*} \cup \{\underline{Fehler}\})). \end{aligned}$$

■

**Konventionen zur Schreibweise:** Um bei der Bezeichnung von Funktionsanwendungen Klammern zu sparen, wird für eine Funktion  $f$  mit Argument  $a$  einfach  $fa$  für die Anwendung von  $f$  auf  $a$  geschrieben. Allerdings wird eine Ausnahme für das erste Argument der Semantikfunktionen gemacht; dieses wird in Doppelklammern gesetzt, um deutlich die syntaktische Einheit, deren Bedeutung bestimmt wird, von den übrigen semantischen Objekten abzuheben.

Ähnlich wie bei den Speicheränderungen in der operationellen Spezifikation, wird die Änderung einer Funktion  $f$  an der Stelle  $x$  mit dem neuen Wert  $a$  durch den

Ausdruck  $f[a/x]$  bezeichnet.

Die Semantikfunktionen lassen sich nun induktiv über den Aufbau der Phrasen angeben:

**Definition 2.16** Sei  $z = (s, e, a) \in ZUSTAND$ .

### 1. Semantik der Terme

$$(a) \quad T[n]z = (n, z) \text{ für alle } n \in ZAHL$$

$$(b) \quad T[x]z = \begin{cases} \underline{\text{Fehler}} & \text{falls } sx = \underline{\text{frei}} \\ (sx, z) & \text{sonst} \end{cases}$$

$$(c) \quad T[T_1 + T_2]z = \begin{cases} \underline{\text{Fehler}} & \text{falls } T[T_1]z = \underline{\text{Fehler}} \text{ oder} \\ & T[T_1]z = (n_1, z_1) \text{ und} \\ & T[T_2]z_1 = \underline{\text{Fehler}} \\ (n_1 + n_2, z_2) & \text{falls } T[T_1]z = (n_1, z_1) \text{ und} \\ & T[T_2]z_1 = (n_2, z_2) \end{cases}$$

(d) analog für alle anderen arithmetischen Operationen

$$(e) \quad T[\text{read}]z = \begin{cases} \underline{\text{Fehler}} & \text{falls } e = \varepsilon \text{ oder } e = b.e' \\ (n, (s, e', a)) & \text{falls } e = n.e' \end{cases}$$

### 2. Semantik der booleschen Terme

$$(a) \quad B[\text{true}]z = (wahr, z)$$

$$(b) \quad B[\text{false}]z = (falsch, z)$$

$$(c) \quad B[\text{not } B]z = \begin{cases} \underline{\text{Fehler}} & \text{falls } B[B]z = \underline{\text{Fehler}} \\ (\neg b, z') & \text{falls } B[B]z = (b, z') \end{cases}$$

$$(d) \quad B[T_1 = T_2]z = \begin{cases} \underline{\text{Fehler}} & \text{falls } T[T_1]z = \underline{\text{Fehler}} \text{ oder} \\ & T[T_1]z = (n_1, z_1) \text{ und} \\ & T[T_2]z_1 = \underline{\text{Fehler}} \\ (n_1 = n_2, z_2) & \text{falls } T[T_1]z = (n_1, z_1) \text{ und} \\ & T[T_2]z_1 = (n_2, z_2) \end{cases}$$

(e) analog für alle anderen booleschen Operationen

$$(f) \quad \mathcal{B}[\text{read}]z = \begin{cases} \underline{\text{Fehler}} & \text{falls } e = \epsilon \text{ oder } e = n.e' \\ (b, (s, e', a)) & \text{falls } e = b.e' \end{cases}$$

### 3. Semantik der Anweisungen

$$(a) \quad \mathcal{C}[\text{skip}]z = z$$

$$(b) \quad \mathcal{C}[I := T]z = \begin{cases} \underline{\text{Fehler}} & \text{falls } T[T]z = \underline{\text{Fehler}} \\ (s[n/I], e', a) & \text{falls } T[T]z = (n, (s, e', a)) \end{cases}$$

$$(c) \quad \mathcal{C}[C_1; C_2]z = \begin{cases} \underline{\text{Fehler}} & \text{falls } \mathcal{C}[C_1]z = \underline{\text{Fehler}} \\ \mathcal{C}[C_1](\mathcal{C}[C_2]z) & \text{sonst} \end{cases}$$

$$(d) \quad \mathcal{C}[\text{if } B \text{ then } C_1 \text{ else } C_2]z = \begin{cases} \underline{\text{Fehler}} & \text{falls } B[B]z = \underline{\text{Fehler}} \\ \mathcal{C}[C_1]z' & \text{falls } B[B]z = (\text{wahr}, z') \\ \mathcal{C}[C_2]z' & \text{falls } B[B]z = (\text{falsch}, z') \end{cases}$$

$$(e) \quad \mathcal{C}[\text{while } B \text{ do } C]z = \begin{cases} \underline{\text{Fehler}} & \text{falls } B[B]z = \underline{\text{Fehler}} \\ \mathcal{C}[C; C]z' & \text{falls } B[B]z = (\text{wahr}, z'), \\ & \text{wobei } C = \text{while } B \text{ do } C \\ z' & \text{falls } B[B]z = (\text{falsch}, z') \end{cases}$$

$$(f) \quad \mathcal{C}[\text{output } T]z = \begin{cases} \underline{\text{Fehler}} & \text{falls } T[T]z = \underline{\text{Fehler}} \\ (s, e', a.n) & \text{falls } T[T]z = (n, (s, e', a)) \end{cases}$$

$$(g) \quad \mathcal{C}[\text{output } B]z = \begin{cases} \underline{\text{Fehler}} & \text{falls } B[B]z = \underline{\text{Fehler}} \\ (s, e', a.b) & \text{falls } B[B]z = (b, (s, e', a)) \end{cases}$$

### 4. Semantik der Programme

$$\mathcal{P}[C]e = \begin{cases} \underline{\text{Fehler}} & \text{falls } \mathcal{C}[C](s_0, e, \epsilon) = \underline{\text{Fehler}} \\ a & \text{falls } \mathcal{C}[C](s_0, e, \epsilon) = (s, e', a), \\ & \text{wobei } s_0 \in \text{SPEICHER} \text{ gegeben ist durch } s_0(I) = \underline{\text{frei}} \text{ für alle } I \in ID. \end{cases}$$

Beachte, daß mit  $\mathcal{P}e$  und  $\mathcal{P}c$  die Definition von  $\mathcal{C}$  rekursiv ist.

Das nächste Beispiel soll den Vorteil verdeutlichen, den die Verwendung der denotationellen Methode gegenüber der operationellen oder Reduktionssemantik beim

Beweisen von Programmeigenschaften bietet. Es beinhaltet den Korrektheitsbeweis zu dem Divisionsprogramm aus Beispiel 2.5, der in Beispiel 2.13 bzgl. der Reduktionssemantik bereits geführt wurde.

### Beispiel 2.17 (Korrektheitsbeweis zu 2.5)

Sei wieder  $C$  das folgende WHILE-Programm:

$$C \left\{ \begin{array}{l} C_1 \left\{ \begin{array}{l} x := \text{read}; \\ y := \text{read}; \\ g := 0; \end{array} \right. \\ \left. \begin{array}{l} \text{while } x \geq y \text{ do} \\ C_2 \left\{ \begin{array}{l} g := g + 1; \\ \left. \begin{array}{l} x := x - y; \\ C_4 \left\{ \begin{array}{l} \text{output } g; \\ \text{output } x \end{array} \right. \end{array} \right. \end{array} \right. \\ C_3 \left\{ \begin{array}{l} \text{output } g; \\ \text{output } x \end{array} \right. \end{array} \right. \end{array} \right.$$

**Voraussetzung:** Seien  $n, m \in \mathbb{N}$  und  $m > 0$ .

**Behauptung:**  $\mathcal{P}[C](n, m) = (q, r)$  mit

$$(*) \quad n = q \cdot m + r \quad \text{und} \quad r < m.$$

**Beweis:** Nach Definition 2.16.4 von  $\mathcal{P}$  genügt es zu zeigen:

$$\mathcal{C}[C](s_0, (n, m), \varepsilon) = (s, \varepsilon, (q, r)) \text{ mit } (*).$$

Aus Definition 2.16.3c, 3b, 1e und 1a folgt:

$$\mathcal{C}[C](s_0, (n, m), \varepsilon) = \mathcal{C}[C_2; C_3](s_0 [0/g] [n/x] [m/y], \varepsilon, \varepsilon).$$

Folgendes Prädikat über dem Zustandsraum entspricht der Aussage  $(*)$ : Sei

$$\begin{aligned} A(z) &:= A_1(z) \wedge A_2(z) \text{ mit} \\ A_1(s, e, a) &:= n = s(g) \cdot m + s(x) \text{ und} \\ A_2(s, e, a) &:= s(x) < s(y). \end{aligned}$$

Nun genügt es wegen der Definition von  $\mathcal{C}[C_3]$ , zu zeigen:

$$A(\mathcal{C}[C_2](s_0 [0/g] [n/x] [m/y], \varepsilon, \varepsilon)).$$

Da  $A_1$  auf dem vorgelegten Zustand gilt, genügt es, zu zeigen:

$$A_1(z) \implies A(\mathcal{C}[C_2]z) \text{ für alle } z = (s, e, a) \in ZUSTAND.$$

Induktion über  $k = s(x)$ :

Induktionsanfang ( $k < s(y)$ )

Aus Definition 2.16.9e folgt  $\mathcal{C}[C_2]z = z$ , und  $A_2(z)$  gilt wegen  $k = s(x)$ .

**Induktionsschritt ( $k \geq s(y)$ )**

*Annahme: Die Behauptung gilt bereits für alle  $z' = (s', e', a')$  mit  $s'(x) < k$ .*

*Aus Definition 2.16.3e und der Tatsache, daß  $s(x) \geq s(y)$  gilt, folgt:*

$$\mathcal{C} [C_2] z = \mathcal{C} [C_4; C_2] z = \mathcal{C} [C_2] z'$$

*mit*

$$z' = (s [s(g) + 1/g] [s(x) - s(y)/x], e, a).$$

*Man prüft leicht nach, daß  $A_1(z')$  gilt, und da wegen  $s(y) = m > 0$  auch  $(s(x) - s(y)) < k$  gilt, folgt die Behauptung aus der Induktionsannahme.*

Q.E.D.

### 2.4.1 Äquivalenz von operationeller und denotationeller Semantik

Zunächst soll eine Schreibweise, in der den Elementen aus dem Zustandskonzept der WSKEA-Maschine ihre äquivalenten Elemente aus dem mathematischen Bereich *ZUSTAND* zugeordnet werden können, vereinbart werden.

Wie gewohnt werden Elemente aus dem Zustandskonzept der operationellen Semantik mit Großbuchstaben und Elemente aus dem Bereich *ZUSTAND* mit Kleinbuchstaben bezeichnet.

Im folgenden sei aber stets  $Z = (S, E, A)$  äquivalent zu  $z = (s, e, a)$ , d.h. für alle  $I \in ID$  gelte

$$s(I) = \begin{cases} k & \text{gdw } S(I) = \underline{k} \\ \underline{frei} & \text{gdw } S(I) = \underline{frei} \end{cases},$$

und ferner sei

$$e = E \text{ und } a = A.$$

Bevor die Äquivalenz der denotationellen und der operationellen Semantik gezeigt wird, soll die Beziehung zwischen operationellen und denotationellen Werten der arithmetischen und der booleschen Ausdrücke hergestellt werden:

#### Lemma 2.18 (Äquivalenz der Semantik der Ausdrücke)

Sei  $(S, E, A)$  eine Konfiguration des Zustandskonzeptes und  $(s, e, a) \in ZUSTAND$  dazu äquivalent.

1. 
$$(T, (S, E, A)) \xrightarrow{*} (n, (S, E', A)) \text{ gdw}$$
  

$$T [T](s, e, a) = (n, (s, e', a))$$
2. 
$$(B, (S, E, A)) \xrightarrow{*} (b, (S, E', A)) \text{ gdw}$$
  

$$B [B](s, e, a) = (b, (s, e', a))$$

**Beweis:** Einfache Induktion über die Struktur von  $T$  bzw.  $B$ .

Es kann nun die Äquivalenz der operationellen und der denotationellen Semantik der Sprache WHILE gezeigt werden:

**Satz 2.19 (Äquivalenz von operationeller und denotationeller Semantik)**

$$\mathcal{P} = eval$$

**Beweis:**

Es genügt, folgende Äquivalenzen zu zeigen:

1.  $\mathcal{C}[C]z = z'$  gdw  $(C, Z) \xrightarrow{*} (\text{skip}, Z')$  und
2.  $\mathcal{C}[C]z = \text{Fehler}$  gdw  $(C, Z) \xrightarrow{*} (C', Z')$  und  $(C', Z')$  lässt sich nicht weiter mit  $\xrightarrow{*}$  reduzieren.

zu 1. „ $\xrightarrow{*}$ “: Sei  $\mathcal{C}[C]z = z'$ .

Die Behauptung lässt sich durch strukturelle Induktion über den Aufbau von  $C$  zeigen.

**Fallunterscheidung:**

$$C = \text{skip}$$

In diesem Fall gilt  $z' = z$  nach Definition 2.16.3a und wegen der Reflexivität von  $\xrightarrow{*}$  gilt auch  $(\text{skip}, Z) \xrightarrow{*} (\text{skip}, Z)$ .

$$C = I := T$$

Aus der Definition 2.16.3b folgt

$$z' = (s[n/I], e', a) \text{ und } \mathcal{T}[T]z = (n, (s, e', a)).$$

Aus Lemma 2.18.1 folgt  $(T, (S, E, A)) \xrightarrow{*} (n, (S, E', A))$ .

Dann gilt nach Definition 2.7.3b auch  $(C, Z) \xrightarrow{*} (\text{skip}, Z')$ .

$$C = C_1; C_2 \mid \text{if } B \text{ then } C_1 \text{ else } C_2 \mid \text{output } T \mid \text{output } B$$

Als Übung

$$C = \text{while } B \text{ do } C_1$$

Aus der Definition 2.16.3e und 3c folgt:

Es gibt ein  $n \in \mathbb{N}$  mit

$$\begin{aligned} \mathcal{C}[C]z &= \mathcal{C}[C](\mathcal{C}[C_1]^n z) = z' \\ \text{und } \mathcal{B}[B](\mathcal{C}[C_1]^k z) &= \text{wahr} \text{ für alle } k < n \\ \text{und } \mathcal{B}[B](\mathcal{C}[C_1]^n z) &= \text{falsch} \\ \text{und } \mathcal{C}[C_1]^n z &= z'. \end{aligned}$$

Daraus folgt

$$\mathcal{C} \llbracket \underbrace{C_1; \dots; C_1}_n \rrbracket z = z', \text{ und per Induktionsannahme gilt}$$

$$(C_1; \dots; C_1, Z) \xrightarrow{*} (\text{skip}, Z').$$

Aus Lemma 2.18.2 folgt nun auch

$$(C, Z) \xrightarrow{*} (\text{skip}, Z').$$

„ $\Leftarrow$ “ Sei  $(C, Z) \xrightarrow{*} (\text{skip}, Z')$  mit  $Z = (S, E, A)$  und  $Z' = (S', E', A')$ . Wegen der Definition 2.16.3a genügt es, allgemein zu zeigen:

Aus  $(C, Z) \Rightarrow (C', Z')$  folgt  $\mathcal{C} \llbracket C \rrbracket z = \mathcal{C} \llbracket C' \rrbracket z'$ .

Diese Aussage wird durch strukturelle Induktion über den Aufbau von  $C$  bewiesen.

#### Fallunterscheidung:

##### $C = \text{skip}$

Da sich  $(\text{skip}, Z)$  nach Definition 2.7 nicht mit  $\Rightarrow$  reduzieren lässt, ist die Prämisse falsch und damit die Aussage wahr.

##### $C = I := T$

Aus Definition 2.7.3b folgt  $C' = \text{skip}$  und  $S' = S[n/I]$  und  $(T, (S, E, A)) \xrightarrow{*} (n, (S, E', A))$ .

Aus Lemma 2.18.1 folgt  $T \llbracket T \rrbracket (s, e, a) = (n, (s, e', a))$ . Dann gilt nach Definition 2.16.3b und 3a auch

$$\mathcal{C} \llbracket C \rrbracket (s, e, a) = (s[n/I], e', a) = \mathcal{C} \llbracket \text{skip} \rrbracket (s[n/I], e', a).$$

##### $C = C_1; C_2 \mid \text{if } B \text{ then } C_1 \text{ else } C_2 \mid \text{output } T \mid \text{output } B$

Als Übung

##### $C = \text{while } B \text{ do } C_1$

(a) Wenn  $(B, (S, E, A)) \xrightarrow{*} (\text{true}, (S, E', A))$ , dann gilt nach Definition 2.7.3f  $(C', Z') = (C_1; C, (S, E', A))$ .

Aus Lemma 2.18.2 folgt  $B \llbracket B \rrbracket z = (\text{wahr}, (s, e', a))$ , und damit folgt aus der Definition 2.16.3e auch

$$\mathcal{C} \llbracket C \rrbracket z = \mathcal{C} \llbracket C_1; C \rrbracket (s, e', a) = \mathcal{C} \llbracket C' \rrbracket z'.$$

- (b) Wenn  $(B, (S, E, A)) \xrightarrow{*} (\text{false}, (S, E', A))$ , dann gilt nach Definition 2.7.3g  $(C', Z') = (\text{skip}, (S, E', A))$ . Aus Lemma 2.18.2 folgt  $B \llbracket B \rrbracket z = (\text{falsch}, (s, e', a))$ , und damit folgt aus der Definition 2.16.3e auch

$$\mathcal{C} \llbracket C \rrbracket z = (s, e', a) = \mathcal{C} \llbracket \text{skip} \rrbracket (s, e', a) = \mathcal{C} \llbracket C' \rrbracket z'.$$

zu 2. Ein genauer Vergleich der Definitionen von  $\mathcal{C}$  (2.16) und  $\Rightarrow$  (2.7) zeigt die gewünschte Äquivalenz.

Q.E.D.

## 2.5 Axiomatische Semantik der Sprache WHILE

In diesem Abschnitt soll die Semantik der Sprache WHILE nach der von Hoare entwickelten axiomatischen Methode ebenfalls auf einer halbformalen Ebene spezifiziert werden.

Zur Formalisierung der axiomatischen Semantik einer Programmiersprache gehören neben der abstrakten Syntax vier Angaben:

1. Eine Menge logischer Ausdrücke oder Prädikate, genannt *Bedingungen*, die über dem Zustandsraum interpretierbar sind. Im allgemeinen wählt man Ausdrücke der Prädikatenlogik erster Stufe.
2. Zu jeder elementaren Anweisung  $C$  ein *Axiom* bzw. ein *Axiomschema*, bestehend aus einer Vorbedingung  $Q$  und einer Nachbedingung  $R$ . Ein solches Axiom wird gegeben durch die Formel

$$\{Q\} C \{R\}$$

und bezeichnet folgende Aussage: Wenn die Bedingung  $Q$  für einen Zustand  $z$  erfüllt ist und die Ausführung von  $C$  mit Anfangszustand  $z$  im Zustand  $z'$  terminiert, dann gilt die Bedingung  $R$  für den Zustand  $z'$ .

3. Zu jeder zusammengesetzten Anweisung  $C$  mit unmittelbaren Komponenten  $C_1, \dots, C_r$  eine *Schlussregel* der Form

$$\frac{F_1, \dots, F_n}{\{Q\} C \{R\}},$$

zu lesen als: Wenn die Formeln  $F_1$  bis  $F_n$  erfüllt sind, dann gilt auch die Formel  $\{Q\} C \{R\}$ . In der Regel besteht die Folge  $F_1, \dots, F_n$  aus Formeln zu den Komponenten  $C_1, \dots, C_r$ .

4. Eine Menge von *allgemeinen Schlussregeln*, die die Gesetze der zugrundeliegenden Logik zur Herleitung neuer gültiger Formeln aus bereits gegebenen gültigen Formeln ausnutzen.

Da die Behandlung von Ausdrücken mit Nebeneffekten nach der axiomatischen Methode einen besonderen Aufwand erfordert, aber keine neuen Einsichten bringt, soll weiterhin folgende Modifikation der Sprache WHILE betrachtet werden: Anstelle des Ausdruckes `read` in der Menge  $\overline{TERM}$  der Terme und der Menge  $BT$  der booleschen Ausdrücke gibt es eine zusätzliche Anweisung der Form `read I` in der Menge  $COM$  der Anweisungen. Die so modifizierte Sprache wird mit WHILE' und die entsprechenden denotationellen Semantikfunktionen mit  $T'$ ,  $B'$ ,  $C'$  und  $P'$  bezeichnet.

Außerdem wird hier zugunsten der Übersichtlichkeit auf eine explizite Fehlerbehandlung verzichtet, d.h. wenn die Ausführung einer Anweisung  $C$  im Zustand  $z$  einen Fehler hervorruft, dann wird keine nichttriviale Formel  $\{Q\} C \{R\}$  mit der Eigenschaft „ $Q$  gilt für  $z$ “ herleitbar sein.

Es soll nun ein Hoare-Kalkül für die Sprache WHILE' mittels der vier erforderlichen Angaben entwickelt werden:

### 1. Bedingungen (Prädikate) auf dem Zustandsraum

Für die Sprache WHILE' genügt es, Bedingungen in Form von aussagenlogischen Ausdrücken zu formulieren. Dabei geht man von der Menge  $\overline{TERM}$  der Terme aus, die über den Konstanten  $KON \cup \overline{KON}$  und den Variablen  $ID \cup \{\underline{input}, \underline{output}\}$  mit Hilfe der Basisoperationen sowie den Konstruktions- und Selektionsfunktionen für Listen aufgebaut sind. Die Bedingungen sind nun prädikatenlogische Formeln, die in gewohnter Weise induktiv aufgebaut werden:

- (i) Jeder boolesche Term aus  $BT$  ist eine Bedingung.
- (ii) Wenn  $Q$  und  $R$  Bedingungen sind, dann sind auch  $(\neg Q)$ ,  $(Q \vee R)$  und  $(Q \wedge R)$  Bedingungen.

### Beispiel: Der Ausdruck

$$((x = 6 \vee \underline{input} = (1, 2, 3)) \wedge y + 3 = z)$$

ist eine Bedingung über dem Zustandsraum.

Jeder solchen Bedingung lässt sich nun bzgl. eines gegebenen Zustandes  $(s, e, a) \in ZUSTAND$  in natürlicher Weise ein Wahrheitswert zuordnen, nachdem für jede Variable  $I$  der dazugehörige Wert  $s(I)$ , für  $\underline{input}$  das Wertetupel  $e$  und für  $\underline{output}$  das Wertetupel  $a$  eingesetzt wurde. Dabei wird einer Bedingung, die einen fehlerhaften Ausdruck enthält, wie etwa  $4/0 = 1$  oder  $x < 0$  bei einem Zustand mit  $s(x) = \underline{frei}$ , der Wahrheitswert *falsch* zugeordnet.

Auf der Menge der Bedingungen ist eine syntaktische Substitution wie folgt

erklärt: Die Bedingung  $Q[t/x]$  bezeichnet die Bedingung  $Q$ , in der jedes Vorkommen von  $x$  durch den Term  $t$  ersetzt wurde.

## 2. Axiome für die elementaren Anweisungen

Für jede Bedingung  $Q$  gilt:

$$\{Q\} \text{ skip } \{Q\} \quad (\text{A.1})$$

$$\{Q[t/I]\} \ I := T \ {Q} , \quad (\text{A.2})$$

wobei  $t$  der semantische Term ist, den man aus  $T$  gewinnt.

$$\{\neg \underline{\text{eof}} \wedge Q[\pi_1(\underline{\text{input}})/I]\} \ \text{read } I \ \{Q[I.\underline{\text{input}}/\underline{\text{input}}]\} , \quad (\text{A.3})$$

wobei  $\neg \underline{\text{eof}}$  abkürzend steht für die Bedingung  $\underline{\text{input}} = \epsilon$  und  $\pi_1$  die Projektion auf die erste Komponente bezeichnet.

$$\{Q[\underline{\text{output}}.t/\underline{\text{output}}]\} \ \text{output } T \ {Q} , \quad (\text{A.4})$$

wobei wieder  $t$  der semantische Term ist, den man aus  $T$  gewinnt.

## 3. Schlußregeln für die zusammengesetzten Anweisungen

Für beliebige Bedingungen  $Q$ ,  $R$  und  $S$  gilt:

$$\frac{\{Q\} \ C_1 \ {R} , \ {R}\} \ C_2 \ {S}}{\{Q\} \ C_1; C_2 \ {S}} \quad (\text{A.5})$$

$$\frac{\{Q \wedge b\} \ C_1 \ {R} , \ \{Q \wedge \neg b\} \ C_2 \ {R}}{\{Q\} \ \text{if } B \text{ then } C_1 \text{ else } C_2 \ {R}} , \quad (\text{A.6})$$

wobei  $b$  der semantische Ausdruck ist, den man aus  $B$  gewinnt.

$$\frac{\{Q \wedge b\} \ C \ {Q}}{\{Q\} \ \text{while } B \text{ do } C \ {Q \wedge \neg b}} , \quad (\text{A.7})$$

wobei  $b$  der zu  $B$  gehörende semantische Ausdruck ist.

Da die Bedingung  $Q$  vor, während und nach Ausführung der while - Schleife gültig ist, heißt sie auch *Schleifeninvariante*.

## 4. Allgemeine Schlußregeln

Wenn die Bedingung  $Q$  im Sinne der Prädikatenlogik eine andere Bedingung  $R$  impliziert und ebenso die Bedingung  $S$  eine Bedingung  $U$  impliziert, dann gilt:

$$\frac{\{R\} \ C \ {S}}{\{Q\} \ C \ {U}} \quad (\text{A.8})$$

Diese Regel heißt auch *Konsequenzenregel (rule of consequence)*.

Diese vier Angaben begründen nun den Hoare-Kalkül zur Sprache WHILE'. Das formale Beweisen einer Formel mit Hilfe der Axiome und Schlußregeln wird durch die Ableitungsregel „ $\vdash$ “ präzisiert.

**Definition 2.20 ( $\vdash$ , Beweisbarkeit von Formeln im Hoare-Kalkül)**

Eine Formel  $F$  kann im Hoare-Kalkül zur Sprache WHILE' bewiesen werden ( $\vdash F$ ), gdw es eine Folge von Formeln  $F_1, F_2, \dots, F_k$  mit  $k \in \mathbb{N}$  und  $F_k = F$  gibt, für die gilt: Jede Formel  $F_\nu$  ( $1 \leq \nu \leq k$ ) ist entweder ein Axiom aus der Menge A.1 bis A.4, oder es folgt aus der Anwendung einer Schlußregel aus der Menge A.5 bis A.8 auf Formeln aus der Teilfolge  $F_1, \dots, F_{\nu-1}$ . ■

Eine Erläuterung dieser Definition bzw. der Wirkungsweise der Axiome und Schlußregeln erfolgt am besten anhand eines Beispiels. Um gleichzeitig die Überlegenheit der axiomatischen Methode gegenüber der operationellen und denotationellen Methode beim Beweisen von Programmeigenschaften zu demonstrieren, soll noch einmal der Korrektheitsbeweis zu dem Divisionsprogramm durchgeführt werden, das jetzt in WHILE' umgeschrieben wurde.

**Beispiel 2.21 (Korrektheitsbeweis zu 2.5)**

Sei  $C$  das folgende WHILE'-Programm:

$$C \left\{ \begin{array}{l} C_1 \left\{ \begin{array}{l} \text{read } x; \\ \text{read } y; \\ g := 0; \end{array} \right. \\ C_2 \left\{ \begin{array}{l} \text{while } x \geq y \text{ do} \\ C_4 \left\{ \begin{array}{l} g := g + 1; \\ x := x - y; \end{array} \right. \end{array} \right. \\ C_3 \left\{ \begin{array}{l} \text{output } g; \\ \text{output } x \end{array} \right. \end{array} \right.$$

Voraussetzung:

$$\begin{aligned} Q &= m > 0 \wedge \underline{\text{input}} = (n, m) \wedge \underline{\text{output}} = \varepsilon \\ R &= n = g \cdot m + x \wedge x < m \wedge \underline{\text{output}} = (g, x) \end{aligned}$$

Behauptung:

$$\vdash \{Q\}C\{R\}$$

Beweis:

$F_1 :$

$Q$  impliziert  $Q_1$  mit

$$\begin{aligned} Q_1 &= \neg \underline{\text{eof}} \wedge m > 0 \wedge \underline{\text{input}} = (n, m) \wedge \underline{\text{output}} = \varepsilon \wedge \\ &\quad \pi_1(\underline{\text{input}}) = n \end{aligned}$$

- $F_2 : (A.3) \vdash \{Q_1\} \text{ read } x \{Q_2\} \text{ mit}$   
 $Q_2 = m > 0 \wedge x.\underline{\text{input}} = (n, m) \wedge \underline{\text{output}} = \epsilon \wedge x = n$
- $F_3 :$   $Q_2$  impliziert  $Q_3$  mit  
 $Q_3 = \neg \underline{\text{eof}} \wedge m > 0 \wedge \underline{\text{input}} = (m) \wedge \underline{\text{output}} = \epsilon \wedge x = n \wedge \pi_1(\underline{\text{input}}) = m$
- $F_4 : (A.8) \vdash \{Q\} \text{ read } x \{Q_3\} \text{ nach } F_1 - F_3$
- $F_5 : (A.9) \vdash \{Q_3\} \text{ read } y \{Q_4\} \text{ mit}$   
 $Q_4 = m > 0 \wedge y.\underline{\text{input}} = (m) \wedge \underline{\text{output}} = \epsilon \wedge x = n \wedge y = m$
- $F_6 :$   $Q_4$  impliziert  $Q_5$  mit  
 $Q_5 = m > 0 \wedge \underline{\text{output}} = \epsilon \wedge x = n \wedge y = m \wedge 0 = 0$
- $F_7 : (A.8) \vdash \{Q_3\} \text{ read } y \{Q_5\} \text{ nach } F_5 \text{ und } F_6$
- $F_8 : (A.5) \vdash \{Q\} \text{ read } x; \text{ read } y \{Q_5\} \text{ nach } F_4 \text{ und } F_7$
- $F_9 : (A.2) \vdash \{Q_5\} g := 0 \{Q_6\} \text{ mit}$   
 $Q_6 = m > 0 \wedge \underline{\text{output}} = \epsilon \wedge x = n \wedge y = m \wedge g = 0$
- $F_{10} : (A.5) \vdash \{Q\} C_1 \{Q_6\} \text{ nach } F_4, F_7 \text{ und } F_9$
- $F_{11} :$   $Q_6$  impliziert die Schleifeninvariante  $Q_7$  mit  
 $Q_7 = m > 0 \wedge \underline{\text{output}} = \epsilon \wedge y = m \wedge n = g \cdot m + x$
- $F_{12} : (A.8) \vdash \{Q\} C_1 \{Q_7\} \text{ nach } F_{10} \text{ und } F_{11}$
- $F_{13} : (A.2) \vdash \{Q_7 \wedge x \geq y\} g := g + 1 \{Q_8\} \text{ mit}$   
 $Q_8 = m > 0 \wedge \underline{\text{output}} = \epsilon \wedge y = m \wedge n = (g - 1) \cdot m + x \wedge x \geq y$
- $F_{14} : (A.2) \vdash \{Q_8\} x := x - y \{Q_9\} \text{ mit}$   
 $Q_9 = m > 0 \wedge \underline{\text{output}} = \epsilon \wedge y = m \wedge n = (g - 1) \cdot m + (x + y) \wedge (x + y) \geq y$
- $F_{15} :$   $Q_9$  impliziert  $Q_7$
- $F_{16} : (A.8) \vdash \{Q_8\} x := x - y \{Q_7\} \text{ nach } F_{14} \text{ und } F_{15}$
- $F_{17} : (A.5) \vdash \{Q_7 \wedge x \geq y\} C_4 \{Q_7\} \text{ nach } F_{13} \text{ und } F_{16}$
- $F_{18} : (A.7) \vdash \{Q_7\} \text{ while } x \geq y \text{ do } C_4 \{Q_7 \wedge \neg x \geq y\} \text{ nach } F_{17}$
- $F_{19} :$   $Q_7 \wedge \neg x \geq y$  impliziert  $Q_{10}$  mit  
 $Q_{10} = \underline{\text{output}}.g.x = (g, x) \wedge n = g \cdot m + x \wedge x < m$

$F_{20} : (A.8) \vdash \{Q_7\} C_2 \{Q_{10}\}$  nach  $F_{18}$  und  $F_{19}$

$F_{21} : (A.4) \vdash \{Q_{10}\} \text{ output } g \{Q_{11}\}$  mit  
 $Q_{11} = \underline{\text{output}}.x = (g, x) \wedge n = g \cdot m + x \wedge x < m$

$F_{22} : (A.4) \vdash \{Q_{11}\} \text{ output } x \{R\}$

$F_{23} : (A.5) \vdash \{Q_{10}\} C_3 \{R\}$  nach  $F_{21}$  und  $F_{22}$

$F_{24} : (A.5) \vdash \{Q\} C \{R\}$  nach  $F_{12}, F_{20}$  und  $F_{23}$

Q.E.D.

Durch eine geschickte Notation läßt sich ein solcher Beweis für die Formel  $\{Q\}C\{R\}$  erheblich in der Länge abkürzen. Man bildet einfach eine Kette der Form

$$\{Q\}t_1\{Q_1\}t_2 \dots t_{k-1}\{Q_{k-1}\}t_k\{R\},$$

wobei für jede Teilfolge der Form

$$\{Q_{\nu-1}\}t_{\nu}\{Q_{\nu}\}$$

gilt: Entweder ist sie eine gültige Hoare-Regel, d.h.  $t_{\nu}$  ist ein Programmsegment, oder sie ist eine prädikatenlogische Schlußfolgerung, d.h.  $t_{\nu}$  steht für „impliziert“, was auch durch den Doppelpfeil ( $\Rightarrow$ ) notiert wird. Natürlich müssen die verwendeten Programmsegmente, der Reihe nach mit dem Semikolon verbunden, gerade den Programmtext  $C$  ergeben.

Mit dieser Notation wird erreicht, daß die Schlußregeln A.5 und A.8 nicht mehr explizit angewendet werden müssen und daß jede Bedingung des Beweises nur einmal auftritt anstatt zweimal, nämlich sowohl als Nachbedingung einer bestimmten Anweisung als auch als Vorbedingung der nächsten Anweisung.

Das Ergebnis dieser Technik ist nun, daß ein Beweis aus einer einfachen Kette besteht, wobei es für jede while - Schleife und jede Verzweigung gesonderte Beweise gibt, die die Prämisse der Schlußregeln A.6 und A.7 sicherstellen.

Der obige Beweis sieht nun in dieser Notation folgendermaßen aus:

$$\{Q\} \Rightarrow \{Q_1\} \text{ read } x \{Q_2\} \Rightarrow \{Q_3\} \text{ read } y \{Q_4\} \Rightarrow \{Q_5\} g := 0 \{Q_6\}$$

$$\Rightarrow \{Q_7\} C_2 \{Q_7 \wedge \neg x \geq y\} \Rightarrow \{Q_{10}\} \text{ output } g \{Q_{11}\} \text{ output } x \{R\}$$

mit

$$\{Q_7 \wedge x \geq y\} g := g + 1 \{Q_8\} x := x - y \{Q_9\} \Rightarrow \{Q_7\}.$$

### 2.5.1 Beziehung der axiomatischen zur denotationellen und operationellen Semantik

Eine Äquivalenz der axiomatischen zur operationellen und denotationellen Semantik lässt sich in der Regel nicht nachweisen, da die semantischen Aussagen der axiomatischen Methode viel allgemeiner sind. Hier können nicht nur funktionale Abhängigkeiten zwischen Ein- und Ausgabe formuliert werden, sondern auch beliebige Beziehungen zwischen Zuständen vor und nach der Ausführung einer Anweisung. Allerdings lassen sich nicht alle gültigen Formeln im Hoare-Kalkül beweisen. Man kann aber, um eine Beziehung zu anderen Semantikformalisierungen herzustellen, die Korrektheit der Hoare-Regeln nachweisen.

Es sollen zunächst die Begriffe der *Korrektheit (soundness)* und der *Vollständigkeit (completeness)* präzisiert werden, wobei sie exemplarisch für die Sprache WHILE' in Bezug auf ihre denotationelle Semantikspezifikation formalisiert werden.

#### Definition 2.22 ( $\models$ , Gültigkeit von Formeln)

Eine Formel  $F = \{Q\}C\{R\}$  ist gültig ( $\models F$ ), gdw für alle  $z \in ZUSTAND$  gilt:

$$\text{Aus } Q \text{ ist wahr bzgl. } z \text{ folgt } R \text{ ist wahr bzgl. } C'[C]z.$$

■

#### Definition 2.23 (Korrektheit des Hoare-Kalküls)

Der Hoare-Kalkül zur Sprache WHILE' ist korrekt, gdw für alle Formeln  $F$  gilt:

$$\text{Aus } \vdash F \text{ folgt } \models F.$$

■

#### Definition 2.24 (Vollständigkeit des Hoare-Kalküls)

Der Hoare Kalkül zu WHILE' ist vollständig, gdw für alle Formeln  $F$  gilt:

$$\text{Aus } \models F \text{ folgt } \vdash F.$$

■

Es ist klar, daß ein Hoare-Kalkül, der bzgl. der denotationellen Semantik korrekt und vollständig ist, die Semantik äquivalent formalisiert, da dann für alle Anweisungen  $C$  gilt:

$$\vdash \{\underline{\text{input}} = e\} \ C \ \{\underline{\text{output}} = P'[C]e\}.$$

Leider kann man für den Hoare-Kalkül zur Sprache WHILE' nur die Korrektheit zeigen:

**Satz 2.25 (Korrektheit der axiomatischen Semantik)**

*Der Hoare Kalkül zu WHILE' ist korrekt bzgl. der denotationellen Semantik von WHILE'.*

**Beweis:** Es genügt zu zeigen, daß jedes Axiom gültig ist und daß die Anwendung einer Schlußregel auf gültige Formeln wieder eine gültige Formel ergibt.

Sei  $z = (s, e, a) \in ZUSTAND$ .

- (A.1) Aus  $Q$  ist *wahr* bzgl.  $z$  folgt  $Q$  ist *wahr* bzgl.  $\mathcal{C}'[\text{skip}]z$ , da nach Definition 2.16.3a  $\mathcal{C}'[\text{skip}]z = z$  gilt.
- (A.2) Sei  $Q[t/I]$  *wahr* bzgl.  $z$  und sei  $T'[T]z = (n, z)$ . (Diese Annahme ist zulässig, weil die Berechnung von Ausdrücken in WHILE' seiteneffektfrei ist.) Da  $t$  der semantische Term ist, der  $T$  entspricht, gilt: Die Bedingung  $t = n$  ist *wahr* bzgl. des Zustandes  $z$ . Nun läßt sich auch induktiv über den Aufbau der Bedingungen zeigen, daß  $Q$  *wahr* ist bzgl. des Folgezustandes  $(s[n/I], e, a)$ .

(A.3), (A.4), (A.5) und (A.6) als Übung

- (A.7) Sei  $\models \{Q \wedge b\} C\{Q\}$ , d.h. für alle  $z \in ZUSTAND$  gilt: Aus  $Q \wedge b$  ist *wahr* bzgl.  $z$  folgt  $Q$  ist *wahr* bzgl.  $\mathcal{C}'[C]z$ .  
Sei nun  $z_1 \in ZUSTAND$  mit  $Q$  ist *wahr* bzgl.  $z_1$ . Wenn die Ausführung der Anweisung **while**  $B$  **do**  $C$  terminiert, so gibt es eine endliche Folge  $z_1, z_2, \dots, z_n$  von Zuständen für die gilt:
  1.  $z_{\nu+1} = \mathcal{C}'[C]z_\nu$  für alle  $1 \leq \nu < n$ ,
  2.  $Q \wedge b$  ist *wahr* bzgl.  $z_\nu$  für alle  $1 \leq \nu < n$ ,
  3.  $Q \wedge \neg b$  ist *wahr* bzgl.  $z_n$  und
  4.  $\mathcal{C}'[\text{while } B \text{ do } C]z = z_n$ .

Damit ist das Prädikat  $Q \wedge \neg b$  bzgl. des Folgezustandes der **while** - Schleife erfüllt.

- (A.8) Die Korrektheit einer Formel, die durch Anwendung der Konsequenzenregel entstanden ist, ergibt sich unmittelbar.

**Q.E.D.**

Die Frage nach der Vollständigkeit der axiomatischen Semantik ist wegen der Allgemeinheit der Formeln schwerer zu beantworten. Für die hier angegebene axiomatische Semantik von WHILE' läßt sich jedoch zeigen, daß sie bzgl. der entsprechenden denotationellen Semantik nicht vollständig ist. Betrachtet man nämlich die Formel

$$F = \{wahr\}C\{falsch\},$$

so besagt sie : Für jeden Zustand  $z$  gilt: Wenn die Ausführung von  $C$  auf  $z$  in Zustand  $z'$  terminiert, dann gilt die Bedingung *falsch* bzgl. des Zustandes  $z'$ . Da die Bedingung *falsch* auf keinem Zustand erfüllt ist, ist diese Formel genau dann gültig ( $\models F$ ), wenn die Anweisung  $C$  auf keinem Zustand terminiert. Wäre nun der angegebene Hoare-Kalkül vollständig, so würde folgende Aussage gelten:

$$\vdash \{wahr\}C\{falsch\} \text{ gdw } C'[\Box] \text{ ist undefiniert für alle } z \in ZUSTAND.$$

Da die Menge aller im Hoare-Kalkül ableitbaren Formeln rekursiv aufzählbar ist, wäre dann auch die Menge aller Anweisungen  $C$ , für die die Formel  $F$  gilt, rekursiv aufzählbar. Dies steht im Widerspruch zur Unentscheidbarkeit des Halteproblems. Also ist obiger Hoare-Kalkül nicht vollständig.

Eine Möglichkeit, diese Schwierigkeiten zu umgehen und dennoch gewisse Aussagen über die Reichweite eines solchen Kalküls zu machen, besteht darin, den Begriff der Vollständigkeit entsprechend einzuschränken. Dazu sei hier auf Cook verwiesen [27].

## Kapitel 3

# Mathematische Grundlagen

In diesem Kapitel sollen diejenigen mathematischen Hilfsmittel vorgestellt werden, die bei der formalen Entwicklung der denotationellen Semantikspezifikationsmethode benötigt werden. Wie in Abschnitt 2.4 erwähnt, müssen zunächst geeignete semantische Bereiche gefunden werden, in denen möglichst genau die in der Programmierung verwendeten Objekte dargestellt werden können. Insbesondere müssen eindeutige Lösungen zu rekursiven Definitionen existieren. In Abschnitt 3.1 wird die Theorie der vollständigen Halbordnungen behandelt, welche die benötigten Eigenschaften besitzen. In Abschnitt 3.2 wird der getypte  $\lambda$ -Kalkül als Metasprache vorgestellt, der sich in besonderem Maße dazu eignet, Objekte in semantischen Bereichen zu definieren und zu benennen. Schließlich wird in Abschnitt 3.3 eine allgemeine Lösungsmethode für rekursive Bereichsdefinitionen diskutiert. Der Stoff des gesamten Kapitels ist ausschließlich im Hinblick auf die Anwendungen im Rahmen der denotationellen Semantik ausgewählt worden. Zur Behandlung weiterführender Fragestellungen sei hier auf die Literatur verwiesen. Die Theorie der semantischen Bereiche wurde im wesentlichen von Scott entwickelt (siehe dazu die Arbeiten [108], [109] und [110]). In dem Buch von Barendregt [11] findet man eine umfassende Darstellung der Theorie des  $\lambda$ -Kalküls, die auch viele weitere Literaturhinweise enthält. Eine Standardmethode zur Lösung rekursiver Bereichsgleichungen wird in der Arbeit von Smyth und Plotkin [112] vorgestellt.

### 3.1 Theorie der semantischen Bereiche

In diesem Abschnitt wird die mathematische Theorie der semantischen Bereiche behandelt, die bei der Spezifikation der denotationellen Semantik im Stil von Scott und Strachey [111] (siehe auch Stoy [113]) verwendet werden.

Ein *semantischer Bereich (domain)* ist eine Struktur, deren Träger alle Datenobjekte eines bestimmten Typs enthält. Dabei kommen sowohl diskrete (endliche) Datenobjekte vor wie z.B. die natürlichen Zahlen oder die Wahrheitswerte als auch unendliche Datenobjekte wie z.B. die Funktionen über den natürlichen Zahlen.

Da die Berechnung unendlicher Objekte nur über endliche Approximationen erfolgen kann, enthalten die semantischen Bereiche neben den totalen auch partielle Elemente. Der Begriff der *Approximation* wird durch eine Ordnungsrelation „ $\sqsubseteq$ “ formalisiert, die auf jedem semantischen Bereich erklärt ist. Man kann für zwei Datenobjekte  $x$  und  $y$  die Beziehung „ $x \sqsubseteq y$ “ verstehen als „ $x$  approximiert  $y$ “ oder „ $x$  enthält weniger Information als  $y$ “ oder „ $x$  ist weniger definiert als  $y$ “.

### Beispiel 3.1 (Der Bereich der partiellen Funktionen über $\mathbb{N}$ )

Sei  $\mathbb{N} \rightarrow \mathbb{N}$  die Menge der partiellen einstetigen Funktionen über den natürlichen Zahlen.

In diesem Bereich sind die endlichen Datenobjekte gerade die Funktionen mit endlichem Definitionsbereich, und entsprechend sind alle Funktionen mit unendlichem Definitionsbereich unendliche Datenobjekte. Die Relation  $\sqsubseteq$  wird durch die Vereinbarung

$$f \sqsubseteq g \text{ gdw } \text{Graph}(f) \subseteq \text{Graph}(g)$$

definiert.

Das unendliche Datenobjekt  $f : \mathbb{N} \rightarrow \mathbb{N}$  mit

$$f(x) = 2x$$

wird von den endlichen Datenobjekten  $f_n : \mathbb{N} \rightarrow \mathbb{N}$  mit

$$f_n(\nu) = \begin{cases} 2\nu & \text{falls } 0 \leq \nu < n \\ \text{undefiniert} & \text{sonst} \end{cases}$$

approximiert. Nun gilt für alle  $n, m \in \mathbb{N}$  mit  $n \leq m$ :

$$f_n \sqsubseteq f_m \sqsubseteq f$$

□

In jedem semantischen Bereich existiert bzgl. der Relation  $\sqsubseteq$  ein *minimales Element*  $\perp$  (bottom), welches jedes Datenobjekt aus diesem Bereich approximiert. Man kann das Element  $\perp$  also verstehen als dasjenige Datenobjekt, das gar keine Information enthält. In obigem Beispiel etwa ist die nirgends definierte Funktion  $\perp = f_0$  das minimale Element des semantischen Bereiches der einstetigen Funktionen über den natürlichen Zahlen.

Das minimale Element  $\perp$  repräsentiert auch die Semantik einer nichtterminierenden Berechnung, die nicht einmal Zwischenergebnisse liefert. So ist z.B. die Semantik der Anweisung `while true do skip` gleich  $\perp$ , während die Semantik der Anweisung `while true do output 1` gleich derjenigen Funktion ist, die jede Eingabe auf die unendliche Folge von 1'nen abbildet.

Eine geeignete Verwendung des minimalen Elementes ermöglicht es auch, partielle Funktionen zu totalen zu erweitern. In obigem Beispiel etwa betrachtet man den

Bereich der einstelligen Funktionen über  $\text{IN} \cup \{\perp\}$  und erweitert entsprechend die Funktionen  $f_n$  zu totalen Funktionen  $f'_n : \text{IN} \cup \{\perp\} \rightarrow \text{IN} \cup \{\perp\}$  durch

$$f'_n(x) = \begin{cases} 2x & \text{falls } x \in \text{IN} \text{ und } 0 \leq x < n \\ \perp & \text{sonst} \end{cases}$$

Dabei wird die Ordnungsrelation übernommen, d.h.:

$$f' \sqsubseteq g' \text{ gdw } f \sqsubseteq g.$$

Eine wichtige Anforderung an die semantischen Bereiche ist die Existenz eindeutiger Lösungen rekursiver Gleichungssysteme. Wie man bereits an der Beispielsprache WHILE sehen kann, wird die denotationelle Semantik der `while`-Schleife rekursiv definiert. Natürlich sollen die dazu aufgestellten Gleichungen ein eindeutiges Objekt als Semantik bestimmen. Im allgemeinen existieren jedoch zu einer rekursiven Gleichung mehrere Lösungen.

### Beispiel 3.2 (Rekursive Gleichungen mit Lösungsmengen)

1.

$$f(x) = f(x) + 1$$

Während die Lösungsmenge zu dieser Definition bzgl. der totalen arithmetischen Funktionen leer ist, gibt es bzgl. des Bereiches  $\text{IN} \cup \{\perp\} \rightarrow \text{IN} \cup \{\perp\}$  genau eine Lösung, nämlich die überall undefinierte Funktion  $x \mapsto \perp$ .

2.

$$f(x) = \begin{cases} 0 & \text{falls } f(x) = 0 \\ 0 & \text{falls } f(x) \neq 0 \end{cases}$$

Lösungsmenge:  $\{x \mapsto 0, x \mapsto \perp\}$

3.

$$f(x, y) = \begin{cases} y & \text{falls } x = 0 \\ f(f(x, y - 1), f(x - 1, y)) & \text{sonst} \end{cases}$$

Lösungen:

(a)

$$(x, y) \mapsto \begin{cases} y & \text{falls } x = 0 \\ \perp & \text{sonst} \end{cases}$$

(b)

$$(x, y) \mapsto y$$

(c)

$$(x, y) \mapsto \underline{\max}(x, y)$$

- (d) Sei  $g : IN \cup \{\perp\} \rightarrow IN \cup \{\perp\}$  eine Funktion mit  $g(x) \neq 0$  und  $g(g(x)) = g(x)$ , dann ist folgende Funktion in der Lösungsmenge:

$$(x, y) \mapsto \begin{cases} y & \text{falls } x = 0 \\ g(x) & \text{sonst} \end{cases}$$

□

Diese Beispiele zeigen, daß es rekursive Gleichungen gibt, die eine, endlich viele oder unendlich viele Lösungen besitzen.

Für die denotationelle Semantik konstruiert man die semantischen Bereiche derart, daß der bzgl. der Approximationsrelation minimale Fixpunkt einer solchen Gleichung stets existiert und vereinbart diesen als Lösung.

Nach diesen Vorüberlegungen soll der Begriff „semantischer Bereich“ formal definiert werden.

### Definition 3.3 (Semantischer Bereich, cpo)

Eine Struktur  $A = (A, \sqsubseteq_A)$  ist ein semantischer Bereich (cpo) (complete partial order), wenn 1. bis 3. gilt:

1. Die Relation  $\sqsubseteq_A$  ist eine Halbordnung auf  $A$  (genannt Approximationsrelation), d.h.  $\sqsubseteq_A$  ist reflexiv, antisymmetrisch und transitiv. Wenn  $A$  aus dem Kontext ersichtlich ist, wird  $\sqsubseteq_A$  durch  $\sqsubseteq$  abgekürzt.
2. In der Trägermenge  $A$  existiert bzgl. der Relation  $\sqsubseteq$  ein minimales Element  $\perp_A$ , d.h. die Beziehung  $\perp_A \sqsubseteq a$  gilt für alle  $a \in A$ . Auch  $\perp_A$  wird durch  $\perp$  abgekürzt, wenn  $A$  aus dem Kontext hervorgeht.
3. Jede Kette  $K \subseteq A$  besitzt bzgl. der Ordnungsrelation  $\sqsubseteq_A$  eine kleinste obere Schranke  $\sqcup K$  in  $A$ , wobei  $K$  Kette heißt, wenn für je zwei Elemente  $k_1, k_2 \in K$  gilt:  $k_1 \sqsubseteq k_2$  oder  $k_2 \sqsubseteq k_1$ .

■

**Konventionen:** Wenn die Ordnungsrelation eines cpo's aus dem Kontext ersichtlich ist, unterscheidet man den Namen des Trägers nicht mehr von dem Namen des cpo's, d.h. man schreibt z.B.  $a \in A$  anstatt  $a \in \underline{A}$ .

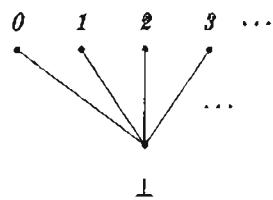
Bildet man den Limes einer Menge von indizierten Elementen, so schreibt man den Laufindex und die Indexmenge auch direkt an das Limessymbol  $\sqcup$  heran, d.h.

$$\bigsqcup_{\nu \in IN} a_\nu \text{ bzw. } \sqcup_{\nu \in IN} a_\nu \text{ steht als Abkürzung für } \sqcup \{a_\nu \mid \nu \in IN\}.$$

Ein cpo lässt sich graphisch veranschaulichen, indem die Elemente des Trägers als Knoten und die Ordnungsrelation durch nach oben gerichtete Kanten dargestellt werden.

#### Beispiel 3.4 (Graphische Darstellung von cpo's )

1. Der semantische Bereich  $\text{IN} \cup \{\perp\}$ :

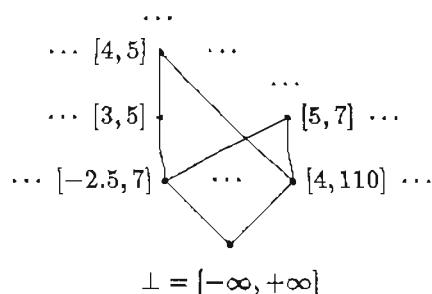


2. Die natürlichen Zahlen mit einem Limeselement  $\infty$  :

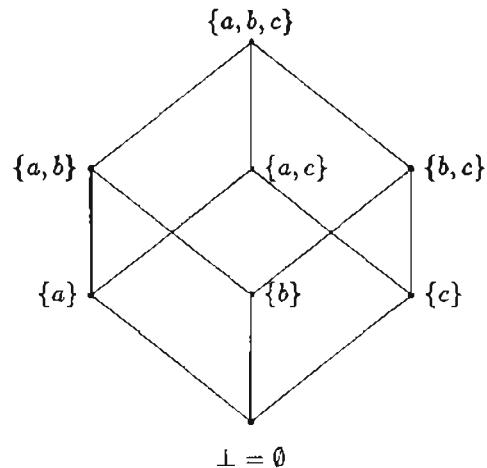


Bemerkung: Ohne Limeselement bilden die natürlichen Zahlen bzgl. der gewohnten kleiner/gleich-Relation kein cpo., da die Bedingung 3. verletzt ist. Z.B. ist die Menge  $\{x \mid x \text{ ist gerade}\}$  eine Kette, die in  $\text{IN}$  keine kleinste obere Schranke besitzt.

3. Die abgeschlossenen Intervalle über den reellen Zahlen mit der Ordnung  $[x_1, y_1] \sqsubseteq [x_2, y_2]$  gdw  $x_1 \leq x_2$  und  $y_1 \geq y_2$  :



4. Die Menge der Teilmengen von  $\{a, b, c\}$  bzgl. der mengentheoretischen Inklusion:



5. Eine halbgeordnete Menge ohne minimales Element, also kein cpo:



□

In der Menge der Funktionen über semantischen Bereichen sollen diejenigen ausgezeichnet werden, die mit dem oben entwickelten Approximationsbegriff verträglich sind. Insbesondere soll von einer Funktion  $f$  gefordert werden, daß sie bei Anwendung auf eine Approximation eines Elementes  $a$  einen Wert liefert, der das Element  $f(a)$  approximiert. Diese Anforderung wird durch den Begriff der *Monotonie* formalisiert. Eine Verschärfung davon verlangt, daß der Wert einer Funktion  $f$ , angewendet auf die kleinste obere Schranke einer Kette  $K$ , auch erhalten werden kann als die kleinste obere Schranke derjenigen Kette, die aus  $K$  entsteht, indem  $f$  auf alle Elemente von  $K$  angewendet wird. Eine Funktion mit dieser Eigenschaft heißt *stetig*. Man kann den Begriff der Stetigkeit auch verstehen als die Eigenschaft, daß eine Funktionsanwendung mit der Bildung der kleinsten oberen Schranke kommutiert.

**Definition 3.5 ( monoton, stetig, strikt und  $[ \longrightarrow ]$  )**  
*Seien  $A$  und  $B$  cpo's und  $f$  eine Funktion von  $A$  nach  $B$ .*

1.  $f$  heißt monoton, wenn für alle  $a_1, a_2 \in A$  gilt:

$$\text{Aus } a_1 \sqsubseteq a_2 \text{ folgt } f(a_1) \sqsubseteq f(a_2).$$

2.  $f$  heißt stetig, wenn für jede Kette  $K \subseteq A$  gilt:  $f(K) := \{f(k) \mid k \in K\}$  ist eine Kette in  $B$  und

$$f(\bigsqcup K) = \bigsqcup f(K).$$

3.  $f$  heißt strikt, wenn

$$f(\perp_A) = \perp_B.$$

4. Die Menge aller stetigen Funktionen von  $A$  nach  $B$  wird mit

$$[A \longrightarrow B]$$

bezeichnet.

■

**Lemma 3.6** Seien  $A$  und  $B$  cpo's.

Wenn  $f : A \longrightarrow B$  stetig ist, dann ist  $f$  auch monoton.

**Beweis:** Seien  $a_1, a_2 \in A$  mit  $a_1 \sqsubseteq a_2$ .

Aus der Tatsache, daß ein Element einer Kette stets weniger definiert ist als die kleinste obere Schranke dieser Kette und aus der Tatsache, daß  $f$  stetig ist, folgt:

$$f(a_1) \sqsubseteq \bigsqcup \{f(a_1), f(a_2)\} = f(\bigsqcup \{a_1, a_2\}) = f(a_2)$$

**Q.E.D.**

Der nun folgende Satz beinhaltet die Existenz des minimalen Fixpunktes stetiger Funktionen sowie eine Aussage darüber, wie ein solcher Fixpunkt als Limes einer Folge von endlichen Approximationen gewonnen werden kann. Mit diesem Satz lassen sich dann eindeutige Lösungen gewisser rekursiver Gleichungen bestimmen, nämlich gerade derjenigen, denen auf natürliche Weise eine stetige Transformation zugeordnet werden kann. Der Fixpunktsatz ist im wesentlichen das Rekursionstheorem von Kleene [61]. Seine Formulierung für vollständige Verbände und cpo's stammt von Tarski [116].

**Satz 3.7 (Fixpunktsatz)**

Sei  $A$  ein cpo, und sei  $f \in [A \longrightarrow A]$ .

Der minimale Fixpunkt von  $f$ ,  $\underline{\text{fix}}(f)$ , existiert in  $A$ , und es gilt:

$$\underline{\text{fix}}(f) = \bigsqcup_{\nu \in \mathbb{N}} f^\nu(\perp).$$

**Beweis:** Die Menge  $\{f^\nu(\perp) \mid \nu \in \mathbb{N}\}$  ist wegen der Minimalität von  $\perp$  und der Monotonie von  $f$  eine Kette. Also existiert ihre kleinste obere Schranke  $\bigcup_{\nu \in \mathbb{N}} f^\nu(\perp)$  in  $A$ . Wegen der Stetigkeit von  $f$  gilt nun:

$$f\left(\bigcup_{\nu \in \mathbb{N}} f^\nu(\perp)\right) = \bigcup_{\nu \in \mathbb{N}} f^{\nu+1}(\perp).$$

Wegen der Minimalität von  $\perp$  gilt:

$$\bigcup_{\nu \in \mathbb{N}} f^{\nu+1}(\perp) = \bigcup_{\nu \in \mathbb{N}} f^\nu(\perp).$$

Also ist  $\bigcup_{\nu \in \mathbb{N}} f^\nu(\perp)$  ein Fixpunkt von  $f$ .

Sei nun  $p$  ein beliebiger Fixpunkt von  $f$ . Aus der Monotonie von  $f$  und der Minimalität von  $\perp$  folgt  $f^\nu(\perp) \sqsubseteq f^\nu(p)$  für alle  $\nu \in \mathbb{N}$ . Also gilt auch:

$$\bigcup_{\nu \in \mathbb{N}} f^\nu(\perp) \sqsubseteq \bigcup_{\nu \in \mathbb{N}} f^\nu(p).$$

Aus der Fixpunkteigenschaft von  $p$  folgt nun

$$\bigcup_{\nu \in \mathbb{N}} f^\nu(p) = p.$$

Damit ist auch die Minimalität von  $\bigcup_{\nu \in \mathbb{N}} f^\nu(\perp)$  gezeigt.

**Q.E.D.**

In den nun folgenden Unterabschnitten sollen einige Konstruktionsmethoden angegeben werden, mit denen sich semantische Bereiche erzeugen lassen.

### 3.1.1 Elementare Bereiche

Aus einer beliebigen, abzählbaren Menge  $M$  wird ein *flacher cpo*

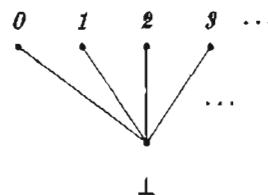
$$M_\perp = (M \cup \{\perp_M\}, \sqsubseteq_M)$$

gebildet, wobei die Ordnungsrelation  $\sqsubseteq_M$  definiert ist durch:

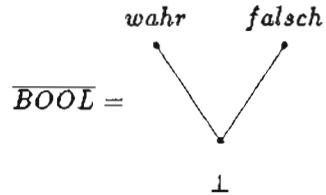
1.  $\perp \sqsubseteq_M m$  für alle  $m \in M \cup \{\perp\}$  und
2.  $m_1 \sqsubseteq_M m_2$  gdw  $m_1 = m_2$  für alle  $m_1, m_2 \in M$ .

#### Beispiel 3.8 (Flache cpo's)

1. Der semantische Bereich  $\mathbb{N}_\perp$  aus Beispiel 3.4.1.:



2. Der semantische Bereich  $\overline{BOOL} = \{\text{wahr}, \text{falsch}\}_{\perp}$ , der booleschen Werte:



3. Der semantische Bereich  $ID$  der Bezeichner:

$$ID = \{I \mid I \text{ ist ein Wort über den kleinen lateinischen Buchstaben}\}_{\perp}$$

□

### 3.1.2 Kartesische Produkte und Folgen

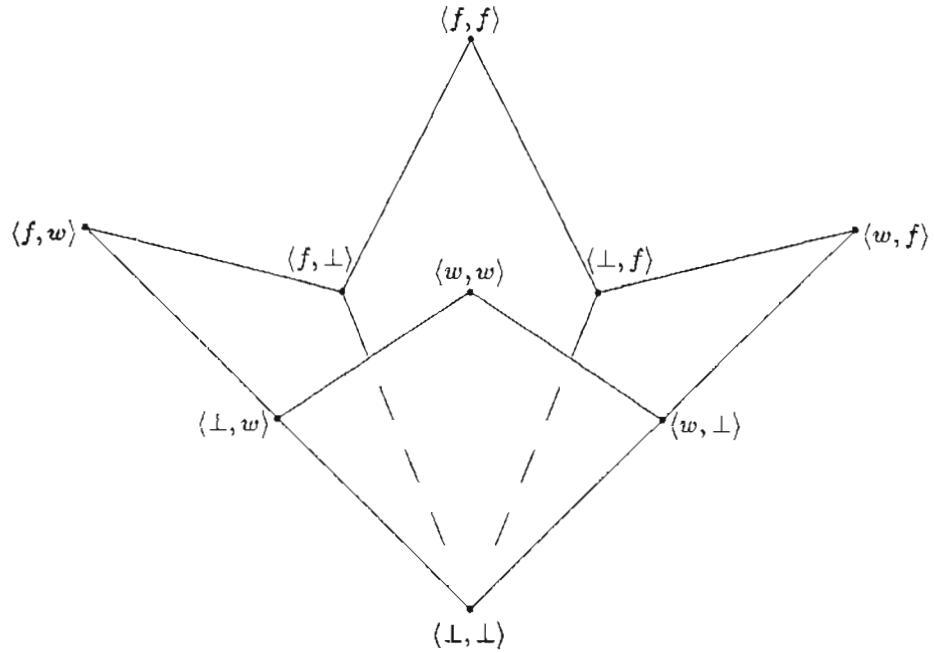
Aus beliebigen semantischen Bereichen  $D_1, D_2, \dots, D_n$  ( $n \in \mathbb{N}$ ) gewinnt man den *kartesischen Produktbereich*

$$D = (D_1 \times D_2 \times \dots \times D_n) = (\{(d_1, \dots, d_n) \mid d_i \in D_i\}, \sqsubseteq_D),$$

wobei hier die Ordnungsrelation  $\sqsubseteq_D$  komponentenweise definiert ist:

$$(d_1, \dots, d_n) \sqsubseteq_D (d'_1, \dots, d'_n) \text{ gdw } d_i \sqsubseteq_{D_i} d'_i \text{ für alle } 1 \leq i \leq n.$$

**Beispiel 3.9 (Der semantische Bereich  $(\overline{BOOL} \times \overline{BOOL})$ )**



□

In analoger Weise ist zu einem semantischen Bereich  $D$  der Bereich

$$D^* = (\{\langle d_i \mid d_i \in D, 1 \leq i \leq n \rangle \mid n \in IN\}, \sqsubseteq_{D^*})$$

der *endlichen Folgen* über  $D$  erklärt, d.h. die Ordnungsrelation ist auch komponentenweise definiert:

$$\begin{aligned} \langle d_i \mid 1 \leq i \leq n \rangle &\sqsubseteq_{D^*} \langle d'_i \mid 1 \leq i \leq m \rangle \text{ gdw} \\ n &\leq m \text{ und } d_i \sqsubseteq_D d'_i \text{ für alle } 1 \leq i \leq n. \end{aligned}$$

Entsprechend ist auch der Bereich

$$D^\omega = (\{\langle d_i \mid d_i \in D, i \in IN \rangle\}, \sqsubseteq_{D^\omega})$$

der *unendlichen Folgen* über  $D$  erklärt mit

$$\langle d_i \mid i \in IN \rangle \sqsubseteq_{D^\omega} \langle d'_i \mid i \in IN \rangle \text{ gdw } d_i \sqsubseteq_D d'_i \text{ für alle } i \in IN.$$

Auf den kartesischen Produktbereichen und den Folgenbereichen sind in natürlicher Weise die *Projektionsfunktionen* erklärt:

$$\begin{aligned} \pi_i : (D_1 \times \cdots \times D_n) &\longrightarrow D_i \\ \langle d_1, \dots, d_n \rangle &\mapsto d_i && \text{für } 1 \leq i \leq n, \text{ bzw.} \\ \pi_i : D^* &\longrightarrow D \\ \langle d_\nu \mid 1 \leq \nu \leq n \rangle &\mapsto \begin{cases} d_i & \text{falls } 1 < i \leq n \\ \perp_D & \text{sonst} \end{cases} && \text{und} \\ \pi_i : D^\omega &\longrightarrow D \\ \langle d_\nu \mid \nu \in IN \rangle &\mapsto d_i && \text{für alle } i \in IN. \end{aligned}$$

Ferner definiert man auf den Folgenbereichen die *Listenoperationen*:

$$\begin{aligned} hd : D^* &\longrightarrow D \\ \langle d_\nu \mid 1 \leq \nu \leq n \rangle &\mapsto \begin{cases} d_1 & \text{falls } n \geq 1 \\ \perp_D & \text{falls } n = 0 \end{cases} \\ tl : D^* &\longrightarrow D^* \\ \langle d_\nu \mid 1 \leq \nu \leq n \rangle &\mapsto \begin{cases} \langle d_{\nu+1} \mid 1 \leq \nu \leq n-1 \rangle & \text{falls } n \geq 1 \\ \perp_{D^*} & \text{falls } n = 0 \end{cases} \\ \underline{null} : D^* &\longrightarrow \overline{BOOL} \\ \langle d_\nu \mid 1 \leq \nu \leq n \rangle &\mapsto \begin{cases} \text{wahr} & \text{falls } n = 0 \\ \text{falsch} & \text{falls } n > 0 \end{cases} \\ \underline{length} : D^* &\longrightarrow IN_\perp \\ \langle d_\nu \mid 1 \leq \nu \leq n \rangle &\mapsto n \end{aligned}$$

$$\begin{aligned}\underline{\text{cons}} : (D \times D^*) &\longrightarrow D^* \\ (d, \langle d_\nu \mid 1 \leq \nu \leq n \rangle) &\mapsto \langle d, d_1, \dots, d_n \rangle\end{aligned}$$

$$\begin{aligned}\underline{\text{cons}'} : (D^* \times D) &\longrightarrow D^* \\ (\langle d_\nu \mid 1 \leq \nu \leq n \rangle, d) &\mapsto \langle d_1, \dots, d_n, d \rangle\end{aligned}$$

Die Funktionen  $\underline{hd}$ ,  $\underline{tl}$  und  $\underline{\text{cons}}$  sind in analoger Weise auch auf unendlichen Listen definiert:

$$\begin{aligned}\underline{hd} : D^\omega &\longrightarrow D \\ \langle d_\nu \mid \nu \in \mathbb{N} \rangle &\mapsto d_1\end{aligned}$$

$$\begin{aligned}\underline{tl} : D^\omega &\longrightarrow D^\omega \\ \langle d_\nu \mid \nu \in \mathbb{N} \rangle &\mapsto \langle d_{\nu+1} \mid \nu \in \mathbb{N} \rangle\end{aligned}$$

$$\begin{aligned}\underline{\text{cons}} : (D \times D^\omega) &\longrightarrow D^\omega \\ (d, \langle d_\nu \mid \nu \in \mathbb{N} \rangle) &\mapsto \langle d, d_1, d_2, \dots \rangle\end{aligned}$$

Die beiden Abbildungen  $\underline{\text{cons}}$  und  $\underline{\text{cons}'}$  werden im allgemeinen wie folgt durch die Punktschreibweise abgekürzt:

Für  $d \in D$  und  $\langle d_1, \dots, d_n \rangle \in D^*$  schreibt man

$d. \langle d_1, \dots, d_n \rangle$  anstelle von  $\underline{\text{cons}}(d, \langle d_1, \dots, d_n \rangle)$  und  
 $\langle d_1, \dots, d_n \rangle.d$  anstelle von  $\underline{\text{cons}'}(\langle d_1, \dots, d_n \rangle, d)$ .

Es ist nun zu zeigen, daß die Produkt- und Folgenbereiche eine cpo-Struktur aufweisen, um sicherzustellen, daß semantische Bereiche erzeugt werden.

### Lemma 3.10 (Erhalt der cpo-Struktur)

Sei  $M$  eine abzählbare Menge, und seien  $D, D_1, \dots, D_n$  cpo's.

1. Die folgenden Bereiche sind cpo's:

- (a)  $M_\perp$ ,
- (b)  $(D_1 \times \dots \times D_n)$ ,
- (c)  $D^*$  und
- (d)  $D^\omega$ .

2. Die Funktionen  $\pi_i$  ( $i \in \mathbb{N}$ ),  $\underline{hd}$ ,  $\underline{tl}$ ,  $\underline{\text{null}}$ ,  $\underline{\text{length}}$ ,  $\underline{\text{cons}}$  und  $\underline{\text{cons}'}$  sind stetig.

Beweis:

1. Es genügt zu zeigen, daß jeweils die drei Bedingungen aus Definition 3.3 erfüllt sind.

(a) Die Bedingungen 1. bis 3. ergeben sich unmittelbar aus der Definition der Struktur  $M_1$ .

- (b)
1. Daß die Relation  $\sqsubseteq_{(D_1 \times \dots \times D_n)}$  eine Halbordnung ist, folgt aus ihrer Definition und aus der Tatsache, daß die Relationen  $\sqsubseteq_{D_i}$ , ( $1 \leq i \leq n$ ), Halbordnungen sind.
  2. Das Element  $\langle \perp_{D_1}, \dots, \perp_{D_n} \rangle$  aus  $(D_1 \times \dots \times D_n)$  ist minimal bzgl. der Relation  $\sqsubseteq_{(D_1 \times \dots \times D_n)}$ .
  3. Sei  $K$  eine Kette in  $(D_1 \times \dots \times D_n)$ . Dann sind nach Definition von  $\sqsubseteq_{D_i}$  auch die Mengen  $\{\pi_i(k) \mid k \in K\}$  Ketten in  $D_i$ , ( $1 \leq i \leq n$ ). Damit ist

$$\sqcup K = \langle \sqcup \{\pi_1(k) \mid k \in K\}, \dots, \sqcup \{\pi_n(k) \mid k \in K\} \rangle$$

die kleinste obere Schranke von  $K$ .

(c) und (d) analog zu (b).

2. ergibt sich unmittelbar aus der Definition und den Beweisen zu 1.

Q.E.D.

### 3.1.3 Summen

Aus gegebenen semantischen Bereichen  $D_1, D_2, \dots, D_n$  ( $n \in \mathbb{N}$ ) gewinnt man den *Summenbereich*

$$D = (D_1 + D_2 + \dots + D_n) = (\{(d, i) \mid d \in D_i, 1 \leq i \leq n\}, \sqsubseteq_D)$$

durch Identifikation der  $\perp$ -Elemente und Vereinigung der Approximationsrelationen, d.h. für alle  $i, j \in \{1, \dots, n\}$  gilt:

$$\begin{aligned} \perp_D &= (\perp_{D_i}, i) \quad \text{und} \\ (d, i) \sqsubseteq_D (d', j) &\quad \text{gdw } d = \perp_{D_i} \text{ oder } i = j \text{ und } d \sqsubseteq_{D_i} d'. \end{aligned}$$

Auf den Summenbereichen sind in natürlicher Weise die *Selektionsfunktionen*  $out_i$ , die *Injektionsfunktionen*  $in_i$  und die *Testfunktionen*  $is_i$  für alle  $1 \leq i \leq n$  erklärt:

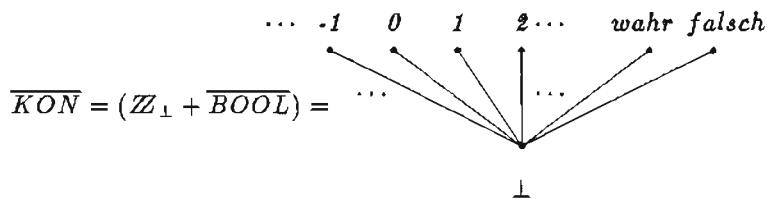
$$\begin{aligned} out_i : (D_1 + \dots + D_n) &\longrightarrow D_i \\ (d, j) &\mapsto \begin{cases} d & \text{falls } i = j \\ \perp_{D_i} & \text{sonst} \end{cases} \end{aligned}$$

$$\begin{aligned} \text{in}_i : D_i &\longrightarrow (D_1 + \cdots + D_n) \\ d &\mapsto (d, i) \end{aligned}$$

$$\begin{aligned} \text{is}_i : (D_1 + \cdots + D_n) &\longrightarrow \overline{\text{BOOL}} \\ (d, i) &\mapsto \begin{cases} \text{wahr} & \text{falls } i = j \text{ und } d \neq \perp \\ \text{falsch} & \text{falls } i \neq j \text{ und } d \neq \perp \\ \perp_{\overline{\text{BOOL}}} & \text{falls } d = \perp \end{cases} \end{aligned}$$

**Konventionen:** Wenn aus dem Kontext ersichtlich ist, ob ein Element  $d$  aus einem semantischen Bereich  $D$  oder aus einem Summenbereich ( $\cdots + D + \cdots$ ) stammt, so wird im allgemeinen nicht zwischen  $d$  und  $(d, i)$  unterschieden, so daß die Projektions- und Injektionsfunktionen weggelassen werden können. Bei einer Summe *disjunkter* Bereiche schreibt man die Testfunktionen auch mit dem jeweiligen Bereichsnamen auf, d.h.  $\text{is}_D$  steht anstelle von  $\text{is}_i$ .

### Beispiel 3.11 (Der Summenbereich ( $\mathbb{Z}_{\perp} + \overline{\text{BOOL}}$ ))



$$\begin{aligned} \text{is}_{\mathbb{Z}_{\perp}}(\text{wahr}) &= \text{falsch} \\ \text{is}_{\overline{\text{BOOL}}}(\text{falsch}) &= \text{wahr} \end{aligned}$$

□

Das folgende Lemma beinhaltet den Erhalt der cpo-Struktur bei der Summenkonstruktion.

### Lemma 3.12

Sei  $n \in \mathbb{N}$  und seien  $D_1, D_2, \dots, D_n$  cpo's.

1.  $D = (D_1 + \cdots + D_n)$  ist ein semantischer Bereich.
2. Die Funktionen  $\text{out}_i$ ,  $\text{in}_i$  und  $\text{is}_i$  sind stetig für alle  $1 \leq i \leq n$ .

### Beweis:

1. Es ist wieder zu zeigen, daß die drei Bedingungen aus Definition 3.3 erfüllt sind.

1. Daß die Relation  $\sqsubseteq_D$  eine Halbordnung ist, folgt aus ihrer Definition und aus der Tatsache, daß die Relationen  $\sqsubseteq_{D_i}$ , ( $1 \leq i \leq n$ ), Halbordnungen sind.
2. Das Element  $\perp_D = (\perp_{D_1}, 1) = \dots = (\perp_{D_n}, n)$  aus  $D$  ist minimal bzgl. der Relation  $\sqsubseteq_D$ .
3. Sei  $K$  eine Kette in  $D$ . Dann gibt es nach Definition von  $\sqsubseteq_D$  ein  $i \in \{1, \dots, n\}$  mit  $out_i(K) = \{out_i(k) \mid k \in K\}$  ist eine Kette in dem Bereich  $D_i$ . Also existiert

$$in_i(\bigsqcup\{out_i(k) \mid k \in K\}) ,$$

die kleinste obere Schranke von  $K$  in  $D$ .

2. ergibt sich unmittelbar aus der Definition der Ordnungsrelation auf  $(D_1 + \dots + D_n)$ .

**Q.E.D.**

### 3.1.4 Funktionen

Zu zwei semantischen Bereichen  $D_1$  und  $D_2$  ist der *Funktionenbereich*

$$D = [D_1 \longrightarrow D_2] = (\{f : D_1 \longrightarrow D_2 \mid f \text{ ist stetig}\}, \sqsubseteq_D)$$

durch eine *punktweise* Ordnungsrelation erklärt, d.h. für alle stetigen Funktionen  $f : D_1 \longrightarrow D_2$  und  $g : D_1 \longrightarrow D_2$  gilt:

$$f \sqsubseteq_D g \text{ gdw } f(x) \sqsubseteq_{D_2} g(x) \text{ für alle } x \in D_1 .$$

#### Lemma 3.13

Seien  $D_1$  und  $D_2$  cpo's. Der Funktionenbereich  $D = [D_1 \longrightarrow D_2]$  ist ein semantischer Bereich.

**Beweis:** Es sind wieder die drei Bedingungen aus Definition 3.3 zu zeigen:

1. Daß die Relation  $\sqsubseteq_D$  eine Halbordnung ist, folgt aus ihrer Definition und aus der Tatsache, daß die Relation  $\sqsubseteq_{D_2}$  eine Halbordnung ist.
2. Das Element  $\perp_D = x \mapsto \perp_{D_2}$  aus  $D$  ist minimal bzgl. der Relation  $\sqsubseteq_D$ .

3. Sei  $K$  eine Kette in  $D$ .

Aus der Definition der Ordnungsrelation  $\sqsubseteq_D$  folgt, daß für alle  $d \in D_1$  die Menge  $\{k(d) \mid k \in K\}$  eine Kette in  $D_2$  ist.

Definiere  $f : D_1 \rightarrow D_2$  durch  $d \mapsto \bigcup K(d) = \bigcup \{k(d) \mid k \in K\}$ . Damit gilt

$$f = \bigcup K.$$

Es bleibt zu zeigen, daß  $f$  stetig ist, also  $f \in D$  gilt.

Sei  $K_1$  eine Kette in  $D_1$ . Für alle  $k \in K$  und  $k_1 \in K_1$  gilt nach Definition von  $\sqcup$ :

$$k(k_1) \sqsubseteq \bigcup K(k_1).$$

Da die Funktion  $k$  nach Lemma 3.6 monoton ist gilt auch:

$$k(k_1) \sqsubseteq k(\bigcup K_1).$$

Daraus folgt:

$$\bigcup \{k(k_1) \mid k_1 \in K_1\} \sqsubseteq \bigcup \{\bigcup K(k_1) \mid k_1 \in K_1\} \text{ und}$$

$$\bigcup \{k(k_1) \mid k \in K\} \sqsubseteq \bigcup \{k(\bigcup K_1) \mid k \in K\}.$$

Da dies für alle  $k_1 \in K_1$  bzw.  $k \in K$  gilt, muß es auch für die entsprechenden kleinsten oberen Schranken gelten:

$$\bigcup \{\bigcup \{k(k_1) \mid k_1 \in K_1\} \mid k \in K\} \sqsubseteq \bigcup \{\bigcup K(k_1) \mid k_1 \in K_1\} \text{ und}$$

$$\bigcup \{\bigcup \{k(k_1) \mid k \in K\} \mid k_1 \in K_1\} \sqsubseteq \bigcup \{k(\bigcup K_1) \mid k \in K\}.$$

Aus der Stetigkeit von  $k$  folgt nun, daß die linke Seite der ersten Ungleichung gleich der rechten Seite der zweiten Ungleichung ist, und daß die linke Seite der zweiten Ungleichung gleich der rechten Seite der ersten Ungleichung ist. Daraus ergibt sich dann auch folgende Gleichung:

$$\bigcup \{k(\bigcup K_1) \mid k \in K\} = \bigcup \{\bigcup K(k_1) \mid k_1 \in K_1\}.$$

Nun gilt:

$$\begin{aligned} f(\bigcup K_1) &= \bigcup \{k(\bigcup K_1) \mid k \in K\} \quad \text{nach Definition von } f \\ &= \bigcup \{\bigcup K(k_1) \mid k_1 \in K_1\} \quad \text{nach obiger Gleichung} \\ &= \bigcup \{f(k_1) \mid k_1 \in K_1\} \quad \text{wegen } f = \bigcup K \\ &= \bigcup f(K_1). \end{aligned}$$

Damit ist  $f$  stetig.

Q.E.D.

Das nächste Lemma zeigt, daß einige, häufig verwendete Funktionen stetig sind, d.h. daß sie in den semantischen Funktionsbereichen liegen.

**Lemma 3.14**

Die Klasse der stetigen Funktionen enthält die Identitätsfunktionen und die konstanten Funktionen und ist unter Komposition, Substitution und Limesbildung abgeschlossen, d.h. für alle cpo's  $A, B, B_1, \dots, B_r, C, D, D_1, D_2$  und  $D_3$  ( $r \in \mathbb{N}$ ) sind folgende Operationen stetig:

## 1. Die Identitätsfunktionen

$$\begin{aligned} id_D : D &\longrightarrow D \\ x &\mapsto x, \end{aligned}$$

## 2. die konstanten Funktionen

$$\begin{aligned} k_d : D_1 &\longrightarrow D_2 \\ x &\mapsto d \end{aligned}$$

für alle  $d \in D_2$ ,

## 3. die Komposition

$$\begin{aligned} f \circ g : D_1 &\longrightarrow D_3 \\ x &\mapsto f(g(x)) \end{aligned}$$

für alle  $f \in [D_2 \longrightarrow D_3]$  und  $g \in [D_1 \longrightarrow D_2]$ ,

## 4. die Substitution

$$\begin{aligned} f(g_1 \times \dots \times g_r) : A &\longrightarrow C \\ x &\mapsto f(\langle g_1(x), \dots, g_r(x) \rangle) \end{aligned}$$

für alle  $f \in [(B_1 \times \dots \times B_r) \longrightarrow C]$  und  $g_\nu : A \longrightarrow B_\nu$  mit  $1 \leq \nu \leq r$  und

## 5. die Limesbildung

$$\begin{aligned} \sqcup F : A &\longrightarrow B \\ x &\mapsto \sqcup\{f(x) \mid f \in F\} \end{aligned}$$

für jede Kette  $F \in [A \longrightarrow B]$ .

**Beweis:**

1. bis 4. standard,
5. analog zum Beweis von Lemma 3.13.

**Q.E.D.**

**Konventionen:** Da von hier an nur stetige Funktionen behandelt werden, können die äußeren eckigen Klammern um die Funktionenräume weggelassen werden.

Um weitere Klammern einzusparen, vereinbart man Rechtsassoziativität von  $\rightarrow$ , d.h.

$D_1 \rightarrow D_2 \rightarrow \cdots \rightarrow D_{n-1} \rightarrow D_n$  steht für

$$[D_1 \rightarrow [D_2 \rightarrow \cdots \rightarrow [D_{n-1} \rightarrow D_n] \dots]] .$$

### Beispiel 3.15 (Der Funktionenbereich *SPEICHER*)

$$SPEICHER = [ID \rightarrow (\mathcal{U}_\perp + \{\underline{frei}\}_1)]$$

Typische Funktionen aus diesem Bereich sind:

$$S_0 = x \mapsto \underline{frei}$$

und

$$S = S_0 [1/a] [-1/b] [0/c] = x \mapsto \begin{cases} 1 & \text{falls } x = a \\ -1 & \text{falls } x = b \\ 0 & \text{falls } x = c \\ \underline{frei} & \text{sonst} \end{cases}$$

Bemerkung: Es gilt nicht etwa  $S_0 \sqsubseteq S$ , da beide Elemente total sind. Hingegen gilt für den Speicher

$$S' = x \mapsto \begin{cases} 1 & \text{falls } x = a \\ -1 & \text{falls } x = b \\ \perp & \text{sonst} \end{cases}$$

$$S' \sqsubseteq S .$$

□

### 3.1.5 Rekursiv definierte Bereiche

Sei  $\mathcal{B} = \{B_1, B_2, \dots, B_n\}$  eine Menge von semantischen Bereichen und sei  $\mathcal{D} = \{D_1, D_2, \dots, D_r\}$  eine Menge von Bereichsvariablen. Zunächst ist die Menge  $T[\mathcal{B}, \mathcal{D}]$  von Bereichstermen über  $\mathcal{B}$  und  $\mathcal{D}$  induktiv wie folgt definiert:

1.  $\mathcal{B} \subseteq T[\mathcal{B}, \mathcal{D}]$  und  
 $\mathcal{D} \subseteq T[\mathcal{B}, \mathcal{D}]$ .
2. Seien  $\tau, \tau_1, \dots, \tau_k \in T[\mathcal{B}, \mathcal{D}]$ , dann sind auch folgende Terme in  $T[\mathcal{B}, \mathcal{D}]$ :
  - (a)  $(\tau_1 \times \cdots \times \tau_k)$ ,
  - (b)  $\tau^*$ ,
  - (c)  $\tau^\omega$ ,

- (d)  $(\tau_1 + \dots + \tau_k)$  und
- (e)  $[\tau_1 \longrightarrow \tau_2]$ .

Durch ein Gleichungssystem der Form

$$\begin{aligned} D_1 &= \tau_1 \\ D_2 &= \tau_2 \\ &\vdots \\ D_r &= \tau_r \end{aligned}$$

mit  $\tau_\nu \in T[\mathcal{B}, \mathcal{D}]$  werden die semantischen Bereiche  $D_1, D_2, \dots, D_r$  rekursiv definiert.

### Beispiel 3.16 (Der rekursiv definierte Bereich LISTE)

Die Menge der beliebig geschachtelten Listen über einer Menge von Konstanten, wie sie in LISP verwendet werden, lässt sich auf natürliche Weise rekursiv definieren:

$$\text{LISTE} = (\overline{\text{KON}} + (\text{LISTE} \times \text{LISTE}))$$

Unter anderen sind folgende Elemente aus diesem Bereich:

$$\begin{aligned} L_1 &= a, \\ L_2 &= \langle a, \langle b, c \rangle \rangle, \\ L_3 &= \langle \langle a, b \rangle, c \rangle \text{ und} \\ L_4 &= \langle \langle a, \perp \rangle, c \rangle. \end{aligned}$$

Bemerkung: Während die ersten drei Listen nicht in der Relation  $\sqsubseteq$  stehen, da sie total sind, gilt für die beiden letzten:

$$L_4 \sqsubseteq L_3.$$

□

Natürlich soll auch solchen Gleichungssystemen der minimale Fixpunkt einer stetigen Transformation als Lösung zugeordnet werden. Allerdings gibt es unterschiedliche universelle Bereiche, auf denen entsprechende Transformationen definiert werden können und in denen eindeutige Lösungen existieren.

In Abschnitt 3.3 wird gesondert auf verschiedene Lösungsmethoden solcher Gleichungen eingegangen.

## 3.2 Der getypte $\lambda$ -Kalkül als Metasprache

Nachdem im vorangegangenen Abschnitt alle Bereichskonstruktionsmethoden, die bei der denotationellen Semantikspezifikation Verwendung finden, vorgestellt wurden, sollen nun Techniken zur Bezeichnung von Elementen aus semantischen Bereichen entwickelt werden. Die diskreten Objekte sollen natürlich in gewohnter

Weise explizit benannt werden, d.h. die ganzen Zahlen werden in Dezimalschreibweise notiert, die Wahrheitswerte durch die Namen *wahr* und *falsch* bezeichnet usw. Schwieriger wird es bei den unendlichen Objekten, etwa den Funktionen. Da die semantischen Bereiche induktiv definiert wurden, sind auch Funktionen höheren Typs Elemente aus semantischen Bereichen. Die Semantikfunktionen der Sprache WHILE sind bereits Funktionen höheren Typs, z.B.

$$\mathcal{T} : \text{TERM} \longrightarrow [\text{ZUSTAND} \longrightarrow ((\mathbb{Z}_1 \times \text{ZUSTAND}) + \{\text{Fehler}\}_1)]$$

mit

$$\begin{aligned} \text{ZUSTAND} &= (\text{SPEICHER} \times \text{EINGABE} \times \text{AUSGABE}) \\ \text{SPEICHER} &= \text{ID} \longrightarrow (\mathbb{Z}_1 + \{\text{frei}\}_1). \end{aligned}$$

Die Semantik eines Terms ist also eine Funktion, die als erstes Argument eine Funktion erwartet. Solche Funktionen über Funktionen, in der Mathematik auch *Funktionale* genannt, und ebenso auch Funktionen über Funktionalen usw. sind von höherem funktionalen Typ.

Weitere Beispiele für bereits bekannte Funktionale sind die Familien der *Fixpunktoperatoren*

$$\underline{\text{fix}} : [D \longrightarrow D] \longrightarrow D$$

und der Curry-Operatoren

$$\underline{\text{curry}} : [(A \times B) \longrightarrow C] \longrightarrow [A \longrightarrow [B \longrightarrow C]]$$

für alle Bereiche  $A$ ,  $B$ ,  $C$  und  $D$ .

Eine geeignete Möglichkeit, solche Funktionen endlich darzustellen, bietet der von Church entwickelte  $\lambda$ -Kalkül [25].

Die Grundidee des  $\lambda$ -Kalküls lässt sich wie folgt veranschaulichen: Sei  $A$  ein Ausdruck, in dem neben bekannten Konstanten und Operationssymbolen die Variablen  $x_1, x_2, \dots, x_n$  vorkommen; dann lässt sich diesem Ausdruck in natürlicher Weise eine  $n$ -stellige Funktion zuordnen, nämlich gerade diejenige Abbildung, die einem geeigneten Argumentetupel  $(a_1, a_2, \dots, a_n)$  den Wert von  $A$  bzgl. der Belegung  $(x_1 \mapsto a_1, x_2 \mapsto a_2, \dots, x_n \mapsto a_n)$  zuordnet. Diese Funktion wird im  $\lambda$ -Kalkül durch den Ausdruck

$$\lambda(x_1, x_2, \dots, x_n).A$$

bezeichnet. Die Ausdrücke des  $\lambda$ -Kalküls heißen  *$\lambda$ -Ausdrücke* oder auch  *$\lambda$ -Terme*.

Im getypten  $\lambda$ -Kalkül soll jedem  $\lambda$ -Term  $t$  ein *Typ*  $\tau$  zugeordnet werden (geschrieben als  $t : \tau$ ), wobei  $\tau$  der Name desjenigen Bereiches ist, in dem das von  $t$  bezeichnete Element liegt. Dies ist möglich, wenn die Typen aller vorkommenden Variablen, Konstanten und Funktionen bekannt sind.

Nicht immer wird der Typ jeder Variablen angegeben. Z.B. bezeichnet der  $\lambda$ -Ausdruck  $\lambda x.x$  die Identitätsfunktion über jedem semantischen Bereich. Solchen Termen können *generische Typen* zugeordnet werden; dies sind Namen für semantische Bereiche, die durch Bereichsterme, in denen eine oder mehrere Bereichsvariablen vorkommen, gegeben sind. Die Theorie der generischen Typen soll hier

nicht im Detail behandelt werden, dazu sei auf Damas und Milner [30] verwiesen. Im folgenden wird angenommen, daß die Typen aller Variablen explizit angegeben oder aus dem Kontext ersichtlich sind.

### Definition 3.17 (Getypte $\lambda$ -Ausdrücke)

Sei  $\mathcal{D}$  eine Menge von semantischen Bereichen, die unter  $\times$ ,  $*$ ,  $\omega$ ,  $+$  und  $\rightarrow$  abgeschlossen ist. Sei ferner  $\mathcal{X} = \{\mathcal{X}^D \mid D \in \mathcal{D}\}$  eine Familie getypter Variablen und  $\mathcal{K} = \{\mathcal{K}^D \mid D \in \mathcal{D}\}$  eine Familie getypter Konstanten, die die oben eingeführten Familien von Operationen ( $\pi_i$ ,  $hd$ ,  $tl$ , null, cons, cons', out<sub>i</sub>, in<sub>i</sub>, is<sub>i</sub>, fix und curry) enthalten.

Die Menge  $\mathcal{A}_\lambda[\mathcal{D}, \mathcal{X}, \mathcal{K}]$  der getypten  $\lambda$ -Ausdrücke über  $\mathcal{D}$ ,  $\mathcal{X}$  und  $\mathcal{K}$  (Abkürzung  $\mathcal{A}_\lambda$ , wenn  $\mathcal{D}$ ,  $\mathcal{X}$  und  $\mathcal{K}$  aus dem Kontext ersichtlich sind) ist induktiv durch 1. bis 4. definiert:

#### 1. Atome

$x : D \in \mathcal{A}_\lambda$  für alle  $D \in \mathcal{D}$  und  $x \in \mathcal{X}^D$  und  
 $k : D \in \mathcal{A}_\lambda$  für alle  $D \in \mathcal{D}$  und  $k \in \mathcal{K}^D$ .

#### 2. Tupel

$\langle t_1, t_2, \dots, t_r \rangle : (D_1 \times D_2 \times \dots \times D_r) \in \mathcal{A}_\lambda$ , falls  $t_\nu : D_\nu \in \mathcal{A}_\lambda$  für alle  $1 \leq \nu \leq r$ ,  $r \in \mathbb{N}$ .

#### 3. Applikationen

$(t_1 t_2) : D \in \mathcal{A}_\lambda$ , falls  $t_1 : D' \rightarrow D \in \mathcal{A}_\lambda$  und  $t_2 : D' \in \mathcal{A}_\lambda$ .

#### 4. Abstraktionen

(a) monadisch:

$(\lambda x.t) : [D_1 \rightarrow D_2] \in \mathcal{A}_\lambda$  falls  $x \in \mathcal{X}^{D_1}$  und  $t : D_2 \in \mathcal{A}_\lambda$ , sowie

(b) polyadisch:

$(\lambda(x_1, \dots, x_r).t) : [(D_1 \times \dots \times D_r) \rightarrow D] \in \mathcal{A}_\lambda$  falls  $x_\nu \in \mathcal{X}^{D_\nu}$  für  $1 \leq \nu \leq r$  und  $t : D \in \mathcal{A}_\lambda$ . ■

**Bemerkung:** Die polyadische Abstraktion  $(\lambda(x_1, \dots, x_r).t)$  könnte natürlich durch einen analogen Ausdruck  $(\lambda y.t')$  mit  $y \in \mathcal{X}^{(D_1 \times \dots \times D_r)}$  ersetzt werden. Es ist jedoch von Vorteil, daß in  $t$  die einzelnen Komponenten des Argumentes über  $x_\nu$  direkt angesprochen werden können, während in  $t'$  die entsprechenden Projektionen  $(\pi_\nu y)$  verwendet werden müssen.

Man kann auch nur den allgemeinen Fall der polyadischen Abstraktion zulassen, ohne an Ausdrucksstärke zu verlieren, da das einstellige kartesische Produkt ( $D$ ) isomorph ist zu  $D$ . Es gibt jedoch zwei Gründe für die explizite Anführung der monadischen Abstraktion. Erstens ist der reine  $\lambda$ -Kalkül von Church nur monadisch definiert worden, so daß diese Schreibweise eher

vertraut sein dürfte, und zweitens spart man zusätzliche Klammern, indem man  $\lambda x.t$  scheibt anstelle von  $\lambda(x).t'$ , wobei  $t'$  aus  $t$  durch Substitution von  $\langle x \rangle$  für jedes Vorkommen von  $x$  in  $t$  entsteht.

### Konventionen:

1. Wenn der *Typ eines  $\lambda$ -Ausdruckes* aus dem Kontext ersichtlich ist, wird auf dessen Angabe in der Regel verzichtet.
2. Die *Applikation* ist *linksassoziativ*, d.h.

$t_1 t_2 \dots t_n$  steht für  $(\dots (t_1 t_2) \dots t_n)$ .

3. Das *Semikolon* bedeutet Applikation mit *rechtsassoziativer Wirkung* und ist von niedrigerer Priorität als die Hintereinanderschreibung, d.h.

$t_1 \dots t_n; r_1 \dots r_m; s_1 \dots s_u$  steht für  $(t_1 \dots t_n)((r_1 \dots r_m)(s_1 \dots s_u))$ .

4. Eine *mehrfache Abstraktion* der Form

$$(\lambda x_1.(\lambda x_2. \dots .(\lambda x_n.t) \dots ))$$

kann durch den Ausdruck

$$(\lambda x_1 x_2 \dots x_n.t)$$

abgekürzt werden.

5. Der Gültigkeitsbereich einer *Abstraktion ohne Klammerung* erstreckt sich so weit nach rechts wie möglich, d.h. Abstraktion bindet stärker als Applikation.

$$\begin{array}{lll} \text{Es ist } & (t(\lambda x.fab))4 & \text{gleich } (t\lambda x.fab)4 \\ \text{aber } & (\lambda x.\underline{\text{plus}}(4,x))7 & \text{ungleich } \lambda x.\underline{\text{plus}}(4,x)7 \\ \text{und } & (\lambda x.(t_1(t_2t_3))) & \text{gleich } \lambda x.t_1; t_2t_3. \end{array}$$

6. Die bekannten *zweistelligen Basisoperationen*  $+$ ,  $-$ ,  $*$ ,  $\div$ ,  $\wedge$  und  $\vee$  werden auch in  $\lambda$ -Ausdrücken in Infixnotation geschrieben; dabei binden sie schwächer als Applikation und Abstraktion. Z.B. steht

$$\lambda x.2 + fx \text{ für } \lambda x.\underline{\text{plus}}(2,fx).$$

### Beispiel 3.18 (Getypter $\lambda$ -Ausdruck)

Es soll eine dreistellige Funktion  $f$  definiert werden, die angewendet auf eine Liste  $L$  von ganzen Zahlen, eine zweistellige, arithmetische Operation  $g$  und eine ganze Zahl  $k$ , den Wert von  $g$ , angewendet auf  $k$  und die dritte Komponente von  $L$ , als Ergebnis liefert, d.h. folgende Gleichung soll gelten:

$$f L g k = g(k, \Pi_3 L).$$

Einen vollständig getypten  $\lambda$ -Ausdruck, der streng nach Definition 3.17 aufgebaut ist und diese Funktion realisiert, erhält man folgendermaßen:

Sei  $L \in \mathcal{X}^{\mathbb{Z}_\perp}$ ,  $g \in \mathcal{X}^{[(\mathbb{Z}_\perp \times \mathbb{Z}_\perp) \rightarrow \mathbb{Z}_\perp]}$  und  $k \in \mathcal{K}^{\mathbb{Z}_\perp}$ . Dann sind nach Definition 3.17.1 die Atome

$$\begin{aligned} L &: \mathbb{Z}_\perp^* \\ g &: [(\mathbb{Z}_\perp \times \mathbb{Z}_\perp) \rightarrow \mathbb{Z}_\perp] \text{ und} \\ k &: \mathbb{Z}_\perp \end{aligned}$$

in  $\mathcal{A}_\lambda$ .

Nach der Voraussetzung bzgl.  $\mathcal{K}$  ist auch

$$\pi_3 : [\mathbb{Z}_\perp^* \rightarrow \mathbb{Z}_\perp]$$

in  $\mathcal{A}_\lambda$ . Nun gilt:

$$\begin{aligned} (\pi_3 L) &: \mathbb{Z}_\perp \in \mathcal{A}_\lambda && \text{nach 3.17.3,} \\ (k, (\pi_3 L)) &: (\mathbb{Z}_\perp \times \mathbb{Z}_\perp) \in \mathcal{A}_\lambda && \text{nach 3.17.2,} \\ (g(k, (\pi_3 L))) &: \mathbb{Z}_\perp \in \mathcal{A}_\lambda && \text{nach 3.17.3,} \\ (\lambda k. (g(k, (\pi_3 L)))) &: [\mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp] \in \mathcal{A}_\lambda && \text{nach 3.17.4,} \end{aligned}$$

und ebenfalls nach 3.17.4 gilt:

$$(\lambda g. (\lambda k. (g(k, (\pi_3 L))))) : [[(\mathbb{Z}_\perp \times \mathbb{Z}_\perp) \rightarrow \mathbb{Z}_\perp] \rightarrow [\mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp]] \in \mathcal{A}_\lambda.$$

Schließlich erhält man die gewünschte Funktion:

$$f =$$

$$(\lambda L. (\lambda g. (\lambda k. (g(k, (\pi_3 L)))))) : [\mathbb{Z}_\perp \rightarrow [[(\mathbb{Z}_\perp \times \mathbb{Z}_\perp) \rightarrow \mathbb{Z}_\perp] \rightarrow [\mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp]]]$$

in  $\mathcal{A}_\lambda$ , abermals nach Definition 3.17.4.

Nach den oben vereinbarten Konventionen vereinfacht sich  $f$  nun wie folgt:

1. Verzicht auf Typangabe:

$$f = (\lambda L. (\lambda g. (\lambda k. (g(k, (\pi_3 L))))))$$

2. Einsparung von Applikationsklammern gemäß der Linksassoziativität:

$$f = (\lambda L. (\lambda g. (\lambda k. g(k, \pi_3 L))))$$

3. Abkürzung der Mehrfachabstraktion:

$$f = (\lambda L g k. g(k, \pi_3 L))$$

4. Streichung von äußeren Abstraktionsklammern:

$$f = \lambda L g k. g(k, \pi_3 L)$$

Wendet man nun  $f$  auf die Liste  $\langle 1, \underline{tl}(3), 8, 4 \rangle$ , die Operation plus und die Zahl 2 an, so entsteht der Ausdruck

$$(\lambda Lgk.g(k, \pi_3 L))\langle 1, \underline{tl}(3), 8, 4 \rangle \underline{\text{plus}} 2 .$$

Man beachte, daß durch die Anwendung von  $f$  auf Argumente eine Klammerung der Abstraktion nötig wurde, da sich sonst die Gültigkeitsbereiche von  $L$ ,  $g$  und  $k$  über die gesamte Applikation erstreckt hätte.

Durch Substitution der Argumente für die entsprechenden Variablen und Anwendung bekannter Termsimplifikationen läßt sich der Wert dieses Ausdruckes im  $\lambda$ -Kalkül berechnen:

$$\begin{aligned} (\lambda Lgk.g(k, \pi_3 L))\langle 1, \underline{tl}(3), 8, 4 \rangle \underline{\text{plus}} 2 &= (\lambda gk.g(k, \pi_3 \langle 1, \underline{tl}(3), 8, 4 \rangle)) \underline{\text{plus}} 2 \\ &= (\lambda k.\underline{\text{plus}}(k, \pi_3 \langle 1, \underline{tl}(3), 8, 4 \rangle)) 2 \\ &= (\lambda k.k + \pi_3 \langle 1, \underline{tl}(3), 8, 4 \rangle) 2 \\ &= 2 + \pi_3 \langle 1, \underline{tl}(3), 8, 4 \rangle \\ &= 2 + 8 \\ &= 10 . \end{aligned}$$

Die in den Zeilen 1, 2 und 4 durchgeführte Substitution heißt im  $\lambda$ -Kalkül  $\beta$ -Reduktion und wird in Definition 3.26.2 formal eingeführt.

Der Übergang von Zeile 2 auf Zeile 3 erfolgte nach der Konvention 5.

Wie man in Zeile 5 erkennt, kann die Applikation der Operation  $\pi_3$  auf unausgewertete Argumente (...  $tl(3)$  ...) bereits ausgewertet werden. Diese Auswertung und die Ersetzung des Ausdruckes 2 + 8 durch 10 sind Beispiele für Termsimplifikationen, die im  $\lambda$ -Kalkül durch die  $\delta$ -Reduktion (siehe Definition 3.26.3) formalisiert werden.

Die Wahl der Namen  $L$ ,  $g$  und  $k$  ist willkürlich, solange sie nicht miteinander kollidieren, d.h.

$$f = \lambda M h i . h(i, \pi_3 M)$$

wäre eine äquivalente Formulierung. Eine solche Umbenennung gebundener Variablen wird im  $\lambda$ -Kalkül durch die  $\alpha$ -Konversion formalisiert (siehe Definition 3.26.1).

□

Da in einem  $\lambda$ -Ausdruck Variablen frei vorkommen können, z.B.  $b : BOOL$ , kann man ohne eine vorgegebene Belegung dieser Variablen noch nicht sagen, welches semantische Objekt von einem solchen Ausdruck bezeichnet wird. Daher definiert man die Menge  $\mathcal{U}$  der Umgebungen als die Menge aller typerhaltenden Abbildungen von der Menge der Variablen in die Menge der semantischen Bereiche.

**Definition 3.19 (Umgebungen)**

Seien  $\mathcal{D}$  und  $\mathcal{X}$  wie oben. Die Menge  $\mathcal{U}[\mathcal{D}, \mathcal{X}]$  der Umgebungen bzgl.  $\mathcal{D}$  und  $\mathcal{X}$  ist die Menge aller typerhaltenden Abbildungen von  $\bigcup \mathcal{X}$  nach  $\bigcup \mathcal{D}$ . Dabei heißt eine Abbildung  $\rho : \bigcup \mathcal{X} \longrightarrow \bigcup \mathcal{D}$  typerhaltend, wenn für alle  $x \in \mathcal{X}^D$  mit  $D \in \mathcal{D}$  die Eigenschaft  $\rho(x) \in D$  gilt.

Die modifizierte Umgebung  $\rho[d/x]$  entsteht aus der Umgebung  $\rho$  durch Ersetzen des Wertes von  $x$  durch das Element  $d$ , d.h.

$$\rho[d/x](y) = \begin{cases} d & \text{falls } x = y \\ \rho(y) & \text{sonst} \end{cases}$$

■

**Konvention:** Wenn  $\mathcal{D}$  und  $\mathcal{X}$  aus dem Kontext ersichtlich sind, kürzt man  $\mathcal{U}[\mathcal{D}, \mathcal{X}]$  durch  $\mathcal{U}$  ab.

Nun lässt sich die Semantikfunktion  $\llbracket \cdot \rrbracket$ , die jedem  $\lambda$ -Ausdruck  $t$  bzgl. einer Umgebung  $\rho$  das durch  $t$  bezeichnete semantische Objekt  $\llbracket t \rrbracket_\rho$  zuordnet, explizit angeben.

**Definition 3.20 (Semantik der  $\lambda$ -Terme)**

Seien  $\mathcal{D}$ ,  $\mathcal{X}$  und  $\mathcal{K}$  wie oben. Die Semantik der  $\lambda$ -Ausdrücke wird induktiv durch die Funktion

$$\llbracket \cdot, \mathcal{D} \rrbracket : \mathcal{U}[\mathcal{D}, \mathcal{X}] \longrightarrow \mathcal{A}_\lambda[\mathcal{D}, \mathcal{X}, \mathcal{K}] \longrightarrow \bigcup \mathcal{D}$$

bestimmt mit:

1.  $\llbracket x, \mathcal{D} \rrbracket_\rho = \rho(x)$   
 $\llbracket k, \mathcal{D} \rrbracket_\rho = k$
2.  $\llbracket \langle t_1, \dots, t_r \rangle, \mathcal{D} \rrbracket_\rho = \langle \llbracket t_1, \mathcal{D} \rrbracket_\rho, \dots, \llbracket t_r, \mathcal{D} \rrbracket_\rho \rangle$
3.  $\llbracket (t_1 t_2), \mathcal{D} \rrbracket_\rho = \llbracket t_1, \mathcal{D} \rrbracket_\rho (\llbracket t_2, \mathcal{D} \rrbracket_\rho)$
4.  $\llbracket \lambda x. t, \mathcal{D} \rrbracket_\rho = d \mapsto \llbracket t, \mathcal{D} \rrbracket_\rho [d/x]$   
 $\llbracket \lambda(x_1, \dots, x_r). t, \mathcal{D} \rrbracket_\rho = \langle d_1, \dots, d_r \rangle \mapsto \llbracket t, \mathcal{D} \rrbracket_\rho [d_1/x_1] \cdots [d_r/x_r]$

für alle  $\rho \in \mathcal{U}$ ,  $x, x_1, \dots, x_r \in \bigcup \mathcal{X}$ ,  $k \in \bigcup \mathcal{K}$  und  $t, t_1, \dots, t_r \in \mathcal{A}_\lambda$

■

**Konvention:** Der Ausdruck  $\llbracket \cdot, \mathcal{D} \rrbracket$  wird durch den Ausdruck  $\llbracket \cdot \rrbracket$  abgekürzt, wenn die zugrunde liegenden semantischen Bereiche aus dem Kontext ersichtlich sind.

Es muß nun gezeigt werden, daß der Bildbereich der oben definierten Semantikfunktion wirklich in  $\bigcup \mathcal{D}$  liegt. Da vorausgesetzt wurde, daß  $\mathcal{D}$  unter dem kartesischen Produkt abgeschlossen ist, gilt für die Fälle 1. bis 3. diese Eigenschaft trivialerweise. In Fall 4. wird jedoch eine Funktion als Wert der Semantikfunktion, angewendet auf Abstraktionen, definiert. Da die Familie der semantischen Bereiche  $\mathcal{D}$  unter der Bereichskonstruktion  $\longrightarrow$  abgeschlossen ist, genügt es zu zeigen, daß die angegebenen Funktionen stetig sind.

**Lemma 3.21 (Die Semantik der Abstraktion ist stetig)**

Seien  $\mathcal{D}$ ,  $\mathcal{X}$  und  $\mathcal{K}$  wie oben. Sei ferner  $t \in A_\lambda[\mathcal{D}, \mathcal{X}, \mathcal{K}]$ ,  $D \in \mathcal{D}$ ,  $x \in \mathcal{X}^D$  und  $\rho \in \mathcal{U}[\mathcal{D}, \mathcal{X}]$ .

Wenn  $\llbracket t \rrbracket \rho \in D' \in \bigcup \mathcal{D}$  gilt, so ist auch  $\llbracket \lambda x. t \rrbracket \rho \in [D \longrightarrow D'] \in \bigcup \mathcal{D}$ .

**Beweis:** Sei  $\llbracket t \rrbracket \rho \in D'$ . Nach Definition 3.20.4 ist  $f := \llbracket \lambda x. t \rrbracket \rho$  eine Funktion von  $D$  nach  $D'$ . Es bleibt zu zeigen, daß  $f$  stetig ist.

Sei  $S$  eine Kette in  $D$ . Die Eigenschaft

$$f(\bigsqcup S) = \bigsqcup f(S)$$

läßt sich durch strukturelle Induktion über  $t$  zeigen (vgl. Definition 3.17).

1.  $t = y \in \mathcal{X}^{D'}$

$$\begin{aligned} \llbracket \lambda x. y \rrbracket \rho (\bigsqcup S) &= \llbracket y \rrbracket \rho [\bigsqcup S / x] && \text{nach Definition 3.20.4} \\ &= \begin{cases} \bigsqcup S & \text{falls } y = x \\ \rho(y) & \text{sonst} \end{cases} && \text{nach Definition 3.20.1} \\ &= \bigsqcup \{ \llbracket y \rrbracket \rho [s/x] \mid s \in S \} && \text{nach Definition 3.20.1} \\ &= \bigsqcup \{ \llbracket \lambda x. y \rrbracket \rho (s) \mid s \in S \} && \text{nach Definition 3.20.4} \\ &= \bigsqcup \llbracket \lambda x. y \rrbracket \rho (S) \end{aligned}$$

Der Fall  $t = k \in \mathcal{K}^{D'}$  läuft analog.

2.  $t = \langle t_1, \dots, t_r \rangle$ 

$$\begin{aligned}
& [\lambda x. \langle t_1, \dots, t_r \rangle] \rho (\sqcup S) \\
&= [\langle t_1, \dots, t_r \rangle] \rho [\sqcup S/x] && \text{nach Definition 3.20.4} \\
&= \langle [t_1] \rho [\sqcup S/x], \dots, [t_r] \rho [\sqcup S/x] \rangle && \text{nach Definition 3.20.2} \\
&= \langle [\lambda x. t_1] \rho (\sqcup S), \dots, [\lambda x. t_r] \rho (\sqcup S) \rangle && \text{nach Definition 3.20.4} \\
&= \langle \sqcup \{ [\lambda x. t_1] \rho (s) \mid s \in S \}, \dots, \sqcup \{ [\lambda x. t_r] \rho (s) \mid s \in S \} \rangle && \text{nach Induktionsannahme} \\
&= \sqcup \{ \langle [\lambda x. t_1] \rho (s), \dots, [\lambda x. t_r] \rho (s) \rangle \mid s \in S \} && \text{nach Lemma 3.10.1b} \\
&= \sqcup \{ \langle [t_1] \rho [s/x], \dots, [t_r] \rho [s/x] \rangle \mid s \in S \} && \text{nach Definition 3.20.4} \\
&= \sqcup \{ [\langle t_1, \dots, t_r \rangle] \rho [s/x] \mid s \in S \} && \text{nach Definition 3.20.2} \\
&= \sqcup \{ [\lambda x. \langle t_1, \dots, t_r \rangle] \rho (s) \mid s \in S \} && \text{nach Definition 3.20.4} \\
&= \sqcup [\lambda x. \langle t_1, \dots, t_r \rangle] \rho (S)
\end{aligned}$$

3.  $t = (t_1 t_2)$ 

$$\begin{aligned}
& [\lambda x. (t_1 t_2)] \rho (\sqcup S) \\
&= [(t_1 t_2)] \rho [\sqcup S/x] && \text{nach Definition 3.20.4} \\
&= [t_1] \rho [\sqcup S/x] ([t_2] \rho [\sqcup S/x]) && \text{nach Definition 3.20.3} \\
&= \sqcup \{ [t_1] \rho [s/x] \mid s \in S \} (\sqcup \{ [t_2] \rho [s/x] \mid s \in S \}) && \text{nach Induktionsannahme} \\
&= \sqcup \{ [t_1] \rho [s/x] ([t_2] \rho [s/x]) \mid s \in S \} && \text{siehe Beweis zu Lemma 3.13} \\
&= \sqcup \{ [(t_1 t_2)] \rho [s/x] \mid s \in S \} && \text{nach Definition 3.20.3} \\
&= \sqcup \{ [\lambda x. (t_1 t_2)] \rho (s) \mid s \in S \} && \text{nach Definition 3.20.4} \\
&= \sqcup [\lambda x. (t_1 t_2)] \rho (S)
\end{aligned}$$

4.  $t = (\lambda y. t')$ 

$$\begin{aligned}
& [\lambda x. (\lambda y. t')] \rho (\sqcup S) \\
&= [(\lambda y. t')] \rho [\sqcup S/x] && \text{nach Definition 3.20.4} \\
&= d \mapsto [t'] \rho [\sqcup S/x] [d/y] && \text{nach Definition 3.20.4} \\
&= d \mapsto \begin{cases} [t'] \rho [d/x] & \text{falls } x = y \\ [t'] \rho [d/y] [\sqcup S/x] & \text{falls } x \neq y \end{cases} && \text{nach Definition 3.19}
\end{aligned}$$

$$\begin{aligned}
&= d \mapsto \begin{cases} \llbracket t' \rrbracket \rho[d/x] & \text{falls } x = y \\ \llbracket \lambda x. t' \rrbracket \rho[d/y](\sqcup S) & \text{falls } x \neq y \end{cases} && \text{nach Definition 3.20.4} \\
&= d \mapsto \begin{cases} \llbracket t' \rrbracket \rho[d/x] & \text{falls } x = y \\ \sqcup \{ \llbracket \lambda x. t' \rrbracket \rho[d/y](s) \mid s \in S \} & \text{falls } x \neq y \end{cases} && \text{nach Induktionsannahme} \\
&= d \mapsto \begin{cases} \llbracket t' \rrbracket \rho[d/x] & \text{falls } x = y \\ \sqcup \{ \llbracket t' \rrbracket \rho[d/y][s/x] \mid s \in S \} & \text{falls } x \neq y \end{cases} && \text{nach Definition 3.20.4} \\
&\quad - \int d \mapsto \llbracket t' \rrbracket \rho[d/x] && \text{falls } x = y \\
&\quad - \sqcup \{ d \mapsto \llbracket t' \rrbracket \rho[d/y][s/x] \mid s \in S \} && \text{falls } x \neq y \\
&= \sqcup \{ d \mapsto \llbracket t' \rrbracket \rho[s/x][d/y] \mid s \in S \} && \text{nach Definition 3.19} \\
&= \sqcup \{ \llbracket \lambda y. t' \rrbracket \rho[s/x] \mid s \in S \} && \text{nach Definition 3.20.4} \\
&= \sqcup \{ \llbracket \lambda x. \lambda y. t' \rrbracket \rho(s) \mid s \in S \} && \text{nach Definition 3.20.4} \\
&= \sqcup \llbracket \lambda x. \lambda y. t' \rrbracket \rho(S)
\end{aligned}$$

Der Beweis zu  $\underline{t} = (\lambda(y_1, \dots, y_r).t')$  läuft analog.

Q.E.D.

### Korollar 3.22 (Die Semantik der Mehrfachabstraktion ist stetig)

Seien  $\mathcal{D}, \mathcal{X}, \mathcal{K}, t$  und  $\rho$  wie oben. Sei ferner  $\langle y_1, \dots, y_r \rangle \in \sqcup \mathcal{X}^r$  mit  $r \in IN$ .

Die Funktion  $\llbracket \lambda(y_1, \dots, y_r).t \rrbracket \rho$  ist stetig.

### Satz 3.23

Die Semantikfunktion  $\llbracket \quad \rrbracket$  ist typerhaltend und wohldefiniert.

**Beweis:** Die Aussage des Satzes folgt unmittelbar aus Definition 3.20, Lemma 3.21 und Korollar 3.22, sowie den Abschlußeigenschaften von  $\mathcal{D}$ .

Q.E.D.

Damit ist jetzt mit denselben Techniken, mit denen die denotationelle Semantik von Programmiersprachen formuliert werden soll, bereits die Semantik der Metasprache formalisiert.

Ursprünglich wurde von Church für den  $\lambda$ -Kalkül eine Reduktionssemantik definiert, durch die im wesentlichen das systematische Ersetzen von formalen Parametern durch entsprechende Argumente formalisiert wird.

Bevor nun die bedeutungserhaltenden Umformungen, die  $\alpha$ -, die  $\beta$ -, die  $\delta$ - und die  $\eta$ -Reduktion des  $\lambda$ -Kalküls, angegeben werden, muß noch erklärt werden, was freie und gebundene Variablen vorkommen sind.

**Definition 3.24 (Freie und gebundene Variablen)**

Sei  $t \in \mathcal{A}_\lambda$ .

Die Mengen  $Fr(t)$  der freien Variablen in  $t$  und  $Geb(t)$  der gebundenen Variablen in  $t$  sind induktiv über den Aufbau von  $t$  bestimmt (vgl. Definition 3.17):

$$1. Fr(x) = \{x\}, Geb(x) = \emptyset \text{ für alle } x \in \mathcal{X}^D, D \in \mathcal{D} \text{ und}$$

$$Fr(k) = Geb(k) = \emptyset \text{ für alle } k \in \mathcal{K}^D, D \in \mathcal{D}.$$

$$2. Fr(\langle t_1, t_2, \dots, t_r \rangle) = \bigcup_{1 \leq \nu \leq r} Fr(t_\nu)$$

$$Geb(\langle t_1, t_2, \dots, t_r \rangle) = \bigcup_{1 \leq \nu \leq r} Geb(t_\nu)$$

$$3. Fr(t_1 t_2) = Fr(t_1) \cup Fr(t_2)$$

$$Geb(t_1 t_2) = Geb(t_1) \cup Geb(t_2)$$

$$4. Fr(\lambda x. t) = Fr(t) \setminus \{x\}$$

$$Geb(\lambda x. t) = Geb(t) \cup \{x\}$$

$$Fr(\lambda(x_1, \dots, x_r).t) = Fr(t) \setminus \{x_1, \dots, x_r\}$$

$$Geb(\lambda(x_1, \dots, x_r).t) = Geb(t) \cup \{x_1, \dots, x_r\}$$

Die Menge  $Var(t) := Fr(t) \cup Geb(t)$  ist die Menge der Variablen in  $t$ . ■

Auf der Menge der  $\lambda$ -Ausdrücke ist ein Substitutionsoperator  $\hat{\delta}_s^\tau$  erklärt, der jedes freie Vorkommen einer Variable  $x$  durch den Term  $s$  ersetzt.

**Definition 3.25 (Der Substitutionsoperator  $\$$ )**

Sei  $x \in \mathcal{X}^D, D \in \mathcal{D}$  und  $s : D \in \mathcal{A}_\lambda$ .

Der Substitutionsoperator  $\$$ , angewendet auf die Variable  $x$  und den Term  $s$ , ergibt die Funktion  $\hat{\delta}_s^\tau : \mathcal{A}_\lambda \longrightarrow \mathcal{A}_\lambda$ , die induktiv gegeben ist durch:

$$1. \$_s^x y = \begin{cases} s & \text{falls } x = y \\ y & \text{sonst} \end{cases} \quad \text{für alle } y \in \cup \mathcal{X}$$

$$\$_s^x k = k \quad \text{für alle } k \in \cup \mathcal{K}$$

$$2. \$_s^x \langle t_1, \dots, t_r \rangle = \langle \$_s^x t_1, \dots, \$_s^x t_r \rangle$$

$$3. \$_s^x (t_1 t_2) = (\$_s^x t_1 \$_s^x t_2)$$

$$4. \$_s^x \lambda y. t = \begin{cases} \lambda y. t & \text{falls } x = y \\ \lambda y. \$_s^x t & \text{sonst} \end{cases}$$

$$\$^x \lambda(x_1, \dots, x_r).t = \begin{cases} \lambda(x_1, \dots, x_r).t & \text{falls } x \in \{x_1, \dots, x_r\} \\ \lambda(x_1, \dots, x_r).\$^x_s t & \text{sonst} \end{cases}$$

In analoger Weise ist die simultane Substitution für alle  $x_i \in \mathcal{X}^{D_i}$ ,  $D_i \in \mathcal{D}$  und  $s_i : D_i \in \mathcal{A}_\lambda$  für  $1 \leq i \leq r$ ,  $r \in \mathbb{N}$  erklärt:

Die Funktion  $\$^{\bar{x}_1 \dots \bar{x}_r}_{\bar{s}_1 \dots \bar{s}_r} : \mathcal{A}_\lambda \longrightarrow \mathcal{A}_\lambda$  ist induktiv gegeben durch:

1.  $\$^{\bar{x}_1 \dots \bar{x}_r}_{\bar{s}_1 \dots \bar{s}_r} y = \begin{cases} s_i & \text{falls } x_i = y \text{ für ein } 1 \leq i \leq r \\ y & \text{sonst} \end{cases} \quad \text{für } y \in \cup \mathcal{X}$   
 $\$^{\bar{x}_1 \dots \bar{x}_r}_{\bar{s}_1 \dots \bar{s}_r} k = k$
2.  $\$^{\bar{x}_1 \dots \bar{x}_r}_{\bar{s}_1 \dots \bar{s}_r} \langle t_1, \dots, t_n \rangle = \langle \$^{\bar{x}_1 \dots \bar{x}_r}_{\bar{s}_1 \dots \bar{s}_r} t_1, \dots, \$^{\bar{x}_1 \dots \bar{x}_r}_{\bar{s}_1 \dots \bar{s}_r} t_n \rangle$
3.  $\$^{\bar{x}_1 \dots \bar{x}_r}_{\bar{s}_1 \dots \bar{s}_r} (t_1 t_2) = (\$^{\bar{x}_1 \dots \bar{x}_r}_{\bar{s}_1 \dots \bar{s}_r} t_1 \$^{\bar{x}_1 \dots \bar{x}_r}_{\bar{s}_1 \dots \bar{s}_r} t_2)$
4.  $\$^{\bar{x}_1 \dots \bar{x}_r}_{\bar{s}_1 \dots \bar{s}_r} \lambda y. t = \begin{cases} \lambda y. \$^{\bar{x}_1 \dots \bar{x}_{i-1} \bar{x}_{i+1} \dots \bar{x}_r}_{\bar{s}_1 \dots \bar{s}_{i-1} \bar{s}_{i+1} \dots \bar{s}_r} t & \text{falls } x_i = y \text{ für ein } 1 \leq i \leq r \\ \lambda y. \$^{\bar{x}_1 \dots \bar{x}_r}_{\bar{s}_1 \dots \bar{s}_r} t & \text{sonst} \end{cases}$   
 $\$^{\bar{x}_1 \dots \bar{x}_r}_{\bar{s}_1 \dots \bar{s}_r} \lambda(z_1, \dots, z_n). t$   
 $= \begin{cases} \lambda(z_1, \dots, z_n). t & \text{falls } \{x_1, \dots, x_r\} \subseteq \{z_1, \dots, z_n\} \\ \lambda(z_1, \dots, z_n). \$^{\bar{x}_1' \dots \bar{x}_r'}_{\bar{s}_1' \dots \bar{s}_r'} t & \text{falls } \{x_1, \dots, x_r\} \setminus \{z_1, \dots, z_n\} = \{x_1', \dots, x_{q'}\} \end{cases}$

■

Nun können die Reduktionen des  $\lambda$ -Kalküls erklärt werden.

### Definition 3.26 (Reduktionen des $\lambda$ -Kalküls)

Seien  $x, y, x_1, y_1, \dots, x_r, y_r \in \cup \mathcal{X}$  und  $t, s \in \mathcal{A}_\lambda$ .

#### 1. $\alpha$ -Konversion (Variablenumbenennung)

Jede Abstraktion der Form

$$\lambda x. t \text{ oder } \lambda(x_1, \dots, x_r). t$$

ist ein  $\alpha$ -Redex und kann, falls  $y$  bzw.  $\langle y_1, \dots, y_r \rangle$  vom selben Typ wie  $x$  bzw.  $\langle x_1, \dots, x_r \rangle$  ist, wie folgt reduziert werden:

$$\begin{aligned} \lambda x. t &\xrightarrow{\alpha} \lambda y. \$^x_y t && \text{falls } y \notin \text{Var}(t), \\ \lambda(x_1, \dots, x_r). t &\xrightarrow{\alpha} \lambda(y_1, \dots, y_r). \$^{x_1 \dots x_r}_{y_1 \dots y_r} t && \text{falls } \{y_1, \dots, y_r\} \cap \text{Var}(t) = \emptyset. \end{aligned}$$

2.  $\beta$ -Reduktion (Argumentsubstitution)Jeder  $\lambda$ -Term der Form

$$(\lambda x.t)s \text{ oder } (\lambda(x_1, \dots, x_r).t)(s_1, \dots, s_r)$$

ist ein  $\beta$ -Redex und kann wie folgt reduziert werden:

$$\begin{aligned} (\lambda x.t)s &\xrightarrow{\beta} \$^x_s t \quad \text{falls } Fr(s) \cap Geb(t) = \emptyset, \\ (\lambda(x_1, \dots, x_r).t)(s_1, \dots, s_r) &\xrightarrow{\beta} \$^{x_1 \dots x_r}_{s_1 \dots s_r} t \quad \text{falls } Fr(s_1 \dots s_r) \cap Geb(t) = \emptyset. \end{aligned}$$

3.  $\delta$ -Reduktion (Konstantenreduktion)

(a) Auswertung von Basisoperationen, angewendet auf Konstante, z.B.

$$\underline{\text{plus}}(4,8) \xrightarrow{\delta} 12$$

(b) Auswertung von Kombinatoren (Konstanten höheren Typs), z.B.

$$\begin{aligned} \pi_3(t, s, \underline{\text{mult}}(2, 3), 1) &\xrightarrow{\delta} \underline{\text{mult}}(2, 3) \text{ und} \\ \underline{\text{fix}}\ t &\xrightarrow{\delta} (t(\underline{\text{fix}}\ t)) \end{aligned}$$

4.  $\eta$ -Reduktion (Extensionalität)Jeder  $\lambda$ -Term der Form

$$\lambda x.(t\ x) \text{ oder } \lambda(x_1, \dots, x_r).(t(x_1, \dots, x_r))$$

ist ein  $\eta$ -Redex und kann wie folgt reduziert werden:

$$\begin{aligned} \lambda x.(t\ x) &\xrightarrow{\eta} t \\ \lambda(x_1, \dots, x_r).(t(x_1, \dots, x_r)) &\xrightarrow{\eta} t. \end{aligned}$$

5.  $\lambda$ -Reduktion (Vereinigung von  $\xrightarrow{\alpha}$ ,  $\xrightarrow{\beta}$ ,  $\xrightarrow{\delta}$  und  $\xrightarrow{\eta}$ )Ein  $\lambda$ -Term  $t$  ist auf einen Term  $s$  reduzierbar ( $t \rightarrow s$ ), falls  $t \xrightarrow{\alpha} s$ ,  $t \xrightarrow{\beta} s$ ,  $t \xrightarrow{\delta} s$  oder  $t \xrightarrow{\eta} s$ .Die Relationen  $\xrightarrow{\alpha}$ ,  $\xrightarrow{\beta}$ ,  $\xrightarrow{\delta}$ ,  $\xrightarrow{\eta}$  und  $\rightarrow$  werden auf kanonische Weise auf Teilterme übertragen, d.h. wenn  $t \rightarrow s$  gilt, dann gilt auch  $(t\ r) \rightarrow (s\ r)$ ,  $\lambda x.t \rightarrow \lambda x.s$  usw.Mit  $\rightarrow^*$  bezeichnet man die reflexive, transitive Hülle von  $\rightarrow$ . ■

An dieser Stelle endet die formale Behandlung des getypten  $\lambda$ -Kalküls. Weitere Aspekte sollen jedoch noch angesprochen und einige Literaturhinweise gemacht werden.

Ein wichtiges Resultat, welches sich durch Induktion über den Termaufbau nachweisen lässt, beinhaltet die Korrektheit der Reduktionsregeln bzgl. der denotationalen Semantik:

**Satz 3.27 ( $\lambda$ -Reduktionen sind semantikerhaltend)**

Seien  $t$  und  $s \in A_\lambda$ , dann gilt:

$$\text{Aus } t \xrightarrow{*} s \text{ folgt } \llbracket t \rrbracket = \llbracket s \rrbracket.$$

**Beweis:** Induktion über den Termaufbau.

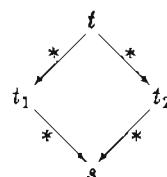
Dieser Satz erlaubt dem Benutzer des  $\lambda$ -Kalküls (etwa als Metasprache) eine uneingeschränkte Anwendung der  $\lambda$ -Reduktionen auf Ausdrücke seiner Sprache. Natürlich hat er dabei das Ziel, seine Ausdrücke so einfach wie möglich darzustellen. Die *Normalformen* bilden in der Menge der  $\lambda$ -Ausdrücke diejenige Teilmenge von Ausdrücken, die nicht mehr durch  $\beta$ -,  $\delta$ - oder  $\eta$ -Reduktionen reduziert werden können, d.h. sie sind bis auf Variablenumbenennung eindeutige minimale Repräsentanten aller auf sie reduzierbaren Ausdrücke.

Es gibt auch sichere Auswertungsstrategien, z.B. die *Standardauswertung*, bei der jeweils der am weitesten links stehende Redex reduziert wird, d.h. wenn ein  $\lambda$ -Ausdruck eine Normalform besitzt, so lässt sich diese mittels einer solchen Strategie erhalten.

In der originalen Version des  $\lambda$ -Kalküls von Church wurde noch keine denotationalen Semantik angegeben, da die benötigten semantischen Bereiche noch nicht bekannt waren. Die Reduktionsemantik von Church ordnet jedem  $\lambda$ -Term seine Normalform als Wert zu, falls eine solche existiert. Terme, die keine Normalform besitzen, werden semantisch als *undefiniert* betrachtet. Die Konsistenz des  $\lambda$ -Kalküls bzgl. der Reduktionssemantik garantiert der *Satz von Church und Rosser*, der besagt, daß zwei Terme, die aus demselben Term hergeleitet wurden, sich stets wieder durch  $\lambda$ -Reduktionen zusammenführen lassen.

**Satz 3.28 (Church und Rosser)**

Seien  $t$ ,  $t_1$  und  $t_2 \in A_\lambda$ , dann gilt: Wenn  $t \xrightarrow{*} t_1$  und  $t \xrightarrow{*} t_2$  gilt, dann gibt es ein  $s \in A_\lambda$  mit  $t_1 \xrightarrow{*} s$  und  $t_2 \xrightarrow{*} s$ , graphisch verdeutlicht durch folgendes Diagramm:



Einen Überblick über die Theorie des  $\lambda$ -Kalküls findet man in dem Buch von Hindley, Lercher und Seldin [47], eine umfassendere Abhandlung beinhaltet das Buch von Curry und Feys [29], und schließlich bietet das ausführliche Buch von Barendregt [11] den aktuellen Stand der Forschung auf diesem Gebiet.

### 3.3 Lösung rekursiver Bereichsgleichungen

Die semantischen Bereiche, die bei der denotationellen Semantikspezifikation Verwendung finden, sind im allgemeinen rekursiv definiert. Dazu zwei Beispiele:

- Der semantische Bereich  $AUS$  von Ausgaben, der aus allen endlichen und unendlichen Folgen von Konstanten aus  $\overline{KON}$  besteht, die mit einem SonderSymbol Fehler oder Stop enden, wird durch folgende Gleichung rekursiv definiert:

$$AUS = \{\text{Fehler}, \text{Stop}\}_\perp + (\overline{KON} \times AUS).$$

- Eine Erweiterung der Sprache WHILE um ein einfaches Prozedurkonzept ohne Parameter entsteht durch die Hinzunahme zweier Anweisungsformen:

$$C ::= \dots | I := \text{proc } C | I | \dots .$$

Die Semantik soll nun derart spezifiziert werden, daß bei Ausführung eines Programms der Form

$$\begin{aligned} & \vdots \\ Q &:= \text{proc } C; \\ & \vdots \\ Q; & \\ & \vdots \\ Q; & \\ & \vdots \end{aligned}$$

jeder Aufruf von  $Q$  eine Ausführung von  $C$  bewirkt.

Dazu definiert man einen geeigneten Bereich zur Modellierung der Semantik solcher Prozeduren,

$$PROC = ZUSTAND \longrightarrow (ZUSTAND + \{\text{Fehler}\}_\perp),$$

und nimmt eine geeignete Erweiterung des Bereiches  $SPEICHER$  vor:

$$SPEICHER = [ID \longrightarrow (\mathbb{Z}_\perp + PROC + \{\text{frei}\}_\perp)].$$

Nun kann man die Semantik der neuen Anweisungen wie folgt erklären:

$$\begin{aligned} \mathcal{C}[I := \text{proc } C](s, e, a) &= (s[\mathcal{C}[C]/I], e, a) \\ \mathcal{C}[I](s, e, a) &= \begin{cases} sI(s, e, a) & \text{falls } \text{is\_PROC}(sI) \\ \text{Fehler} & \text{sonst} \end{cases}. \end{aligned}$$

Das vollständige Gleichungssystem für die semantischen Bereiche lautet nun:

$$\begin{aligned}
 ZUSTAND &= SPEICHER \times EINGABE \times AUSGABE \\
 SPEICHER &= ID \longrightarrow (\mathbb{Z}_\perp + PROC + \{frei\}_\perp) \\
 EINGABE &= \overline{KON}^* \\
 AUSGABE &= \overline{KON}^* \\
 KON &= (\mathbb{Z}_\perp + \overline{BOOL}) \\
 PROC &= ZUSTAND \longrightarrow (ZUSTAND + \{Fehler\}_\perp).
 \end{aligned}$$

Man erkennt, daß der Bereich *ZUSTAND* über den Bereich *SPEICHER*, der Bereich *SPEICHER* über den Bereich *PROC* und der Bereich *PROC* wieder über den Bereich *ZUSTAND* erklärt ist, also ist dieses Gleichungssystem rekursiv.

Zur Lösung solcher rekursiv definierten Bereiche, gibt es prinzipiell zwei Möglichkeiten:

1. Die effektive Konstruktion eines Limesbereiches und
2. die Retraktion aus einem universellen Bereich.

In beiden Fällen erhält man aus einem Gleichungssystem der Form

$$\begin{aligned}
 D_1 &= \tau_1 \\
 D_2 &= \tau_2 \\
 &\vdots \\
 D_r &= \tau_r
 \end{aligned}$$

mit  $\tau_\nu \in T[B, D]$ ,  $1 \leq \nu \leq r$  und  $r \in IN$ , als Lösung semantische Bereiche  $D_1, D_2, \dots, D_r$ , für die gilt:

$$D_\nu \cong \tau_\nu [D_1/D_1] [D_2/D_2] \cdots [D_r/D_r]$$

für alle  $1 \leq \nu \leq r$ . Man begnügt sich also damit, anstelle von Gleichheit Isomorphie zu verwenden, und unterscheidet in der Schreibweise  $D_i$  und  $D_i$  nicht mehr.

Nach der ersten Methode fand Scott 1969 erstmalig ein Modell für den reinen  $\lambda$ -Kalkül, dessen Konstruktion er in der Arbeit [108] beschreibt. Die zweite Methode wurde zunächst von Scott für den universellen Bereich  $P_u$ , der Teilmengen der natürlichen Zahlen, in der Arbeit [109] vorgestellt und später von Smyth und Plotkin [112] verallgemeinert.

Hier soll exemplarisch für eine bestimmte Bereichsgleichung die erste Methode vorgestellt werden, wobei sich die Lösungen anderer Gleichungssysteme in analoger Weise konstruieren lassen. Die zweite Methode erfordert einen erheblich höheren Aufwand an mathematischen Voraussetzungen, daher sei der Leser auf die oben erwähnten Arbeiten sowie auf das Buch von Stoy [113] verwiesen.

Angenommen, man benötigt einen semantischen Bereich  $D$ , der sowohl die Konstanten aus  $\overline{KON}$  als auch alle einstelligen Funktionen und Funktionale beliebiger funktionaler Tiefe über  $\overline{KON}$  enthält. Eine Möglichkeit zur Definition eines solchen Bereiches wäre die Aufstellung der Gleichung

$$D = \overline{KON} + [D \longrightarrow D].$$

Eine andere Möglichkeit wäre die Gleichung

$$D = [D \longrightarrow D].$$

Die erste Definition gestattet es, von jedem Element aus  $D$  mittels der in Abschnitt 3.1.3 eingeführten Tests  $\text{is}_{\overline{KON}}$  und  $\text{is}_{[D \longrightarrow D]}$  die Eigenschaft, Konstante bzw. Funktion zu sein, einfach zu testen, und ist daher für praktische Anwendungen der zweiten vorzuziehen. Zur Illustration der Konstruktionsmethode für Lösungsbereiche, ist die zweite Definition jedoch besser geeignet. Die Gleichung

$$D = [D \longrightarrow D]$$

ist genau diejenige, die jedes Modell des reinen (ungetypten)  $\lambda$ -Kalküls erfüllen muß, da jeder  $\lambda$ -Term sowohl als einstelliger Operator als auch als Operand auftreten kann.

Zur Lösung dieser Gleichung soll die von Scott in [108] vorgestellte Konstruktion durchgeführt werden, wobei jedoch von cpo's anstatt von vollständigen Verbänden als semantische Bereiche ausgegangen wird.

Die Idee ist wie folgt:

Man startet von einem Basisbereich  $D_0$ , der alle atomaren Objekte enthält, z.B.

$$D_0 = \overline{KON}.$$

Induktiv erzeugt man nun alle „endlichen“ Bereiche  $D_{n+1}$  durch Einsetzen von  $D_n$  für  $D$  in der rechten Gleichungsseite. In unserem Fall also

$$D_{n+1} = [D_n \longrightarrow D_n].$$

Schließlich bildet man das unendliche Produkt

$$D_0 \times D_1 \times D_2 \times \dots$$

und zeichnet darin diejenigen Elemente

$$\langle d_i \mid i \in \mathbb{N} \rangle$$

aus, die die Eigenschaft besitzen, daß jede Komponente  $d_i$  eine „Approximation“ aller höheren Komponenten  $d_{i+}$  darstellt. Die Menge dieser Elemente bildet den gesuchten semantischen Bereich  $D$ , bei Scott  $D_\infty$  genannt.

Nun ist jedoch bisher der Begriff der „Approximation“ nur innerhalb eines semantischen Bereiches erklärt worden, nämlich durch die Definition

$$d_1 \text{ approximiert } d_2 \text{ gdw } d_1 \sqsubseteq_D d_2,$$

für alle Bereiche  $D$  und  $d_1, d_2 \in D$ . Es zeigt sich aber, daß sich jeder Bereich  $D_i$  in einen beliebig höheren Bereich  $D_{i+k}$  mittels einer Injektion

$$\varphi^{(i,i+k)} : D_i \longrightarrow D_{i+k}$$

einbetten läßt. Damit kann man die Approximationsrelation  $\sqsubseteq$  durch die Definition

$$d_i \sqsubseteq d_{i+k} \text{ gdw } \varphi^{(i,i+k)}(d_i) \sqsubseteq_{D_{i+k}} d_{i+k},$$

für alle nach obiger Konstruktion erzeugten Bereiche  $D_i$  und  $D_{i+k}$  mit  $d_i \in D_i$  und  $d_{i+k} \in D_{i+k}$ , erweitern.

Neben der Möglichkeit, mittels  $\varphi^{(i,i+k)}$  ein Element in einen höheren Bereich zu injizieren, möchte man auch Elemente aus höheren Bereichen in niedrigere projizieren. Dies wird mittels der Retraktionsfunktionen

$$\psi^{(i+k,i)} : D_{i+k} \longrightarrow D_i$$

geschehen. Jedes Paar

$$(\varphi^{(i,i+k)}, \psi^{(i+k,i)})$$

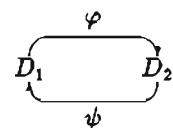
bildet ein Retraktionspaar. Man erwartet natürlich, daß diese Retraktionsfunktionen möglichst viel „Information“ erhalten, d.h. daß man jedes Element aus  $D_i$  nach Anwendung einer Injektion und anschließender Anwendung der entsprechenden Retraktion wieder erhält und umgekehrt, daß das Ergebnis der Anwendung einer Retraktion und anschließender Anwendung der entsprechenden Injektion das Ausgangselement wenigstens approximiert. Diese Eigenschaften werden *Retraktionseigenschaften* genannt.

Formal läuft die Konstruktion wie folgt:

### Definition 3.29 (Retraktionspaar)

Seien  $D_1$  und  $D_2$  semantische Bereiche.

Ein Paar  $(\varphi, \psi)$  von Funktionen



heißt Retraktionspaar (zwischen  $D_1$  und  $D_2$ ), falls  $\varphi$  und  $\psi$  stetig sind und die Retraktionseigenschaften

$$1. \psi \circ \varphi = id_{D_1} \text{ und}$$

$$2. \varphi \circ \psi \sqsubseteq id_{D_2}$$

erfüllen. Dabei heißt  $\varphi$  Injektion und  $\psi$  Retraktion. Der Bereich  $D_1$  heißt Retrakt von  $D_2$  (mittels  $(\varphi, \psi)$ ). ■

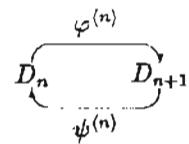
Betrachtet man nun zu dem Bereich  $D_0 := \overline{KON}$  alle Bereiche

$$D_{n+1} := [D_n \longrightarrow D_n],$$

so lassen sich zunächst Retraktionspaare zu je zwei „benachbarten“ Bereichen angeben:

**Definition 3.30 (Retraktionspaare  $(\varphi^{(n)}, \psi^{(n)})$ )**

Sei  $n \in IN$ . Das Retraktionspaar



ist induktiv bestimmt durch:

1.  $\varphi^{(0)}(d) = \lambda x.d$ ,  
 $\psi^{(0)}(f) = f(\perp)$  und
2.  $\varphi^{(n+1)}(d) = \varphi^{(n)} \circ d \circ \psi^{(n)}$ ,  
 $\psi^{(n+1)}(f) = \psi^{(n)} \circ f \circ \varphi^{(n)}$ .

■

**Lemma 3.31** Sei  $n \in IN$ .

Das Paar  $(\varphi^{(n)}, \psi^{(n)})$  ist stetig und erfüllt die Retraktionseigenschaften.

**Beweis:** Die Stetigkeit folgt unmittelbar aus Lemma 3.14. Die Retraktionseigenschaften zeigt man durch Induktion über  $n$ :

$n = 0$  1. Sei  $d \in D_0$ , dann gilt:

$$\begin{aligned} \psi^{(0)} \circ \varphi^{(0)}(d) &= \psi^{(0)}(\varphi^{(0)}(d)) \\ &= (\lambda x.d)\perp && \text{nach Definition von } \varphi^{(0)} \text{ und } \psi^{(0)} \\ &= d && \text{nach Definition 3.26.2 und Satz 3.27.} \end{aligned}$$

2. Sei  $f \in D_1$ , dann gilt:

$$\begin{aligned} \varphi^{(0)} \circ \psi^{(0)}(f) &= \varphi^{(0)}(\psi^{(0)}(f)) \\ &= \lambda x.(f \perp) && \text{nach Definition von } \varphi^{(0)} \text{ und } \psi^{(0)} \\ &\sqsubseteq f && \text{nach Lemma 3.6.} \end{aligned}$$

n+1 1. Sei  $d \in D_{n+1}$ , dann gilt:

$$\begin{aligned}
 & \psi^{(n+1)} \circ \varphi^{(n+1)}(d) \\
 &= \psi^{(n+1)}(\varphi^{(n+1)}(d)) \\
 &= \psi^{(n+1)} \circ \varphi^{(n)} \circ d \circ \psi^{(n+1)} \circ \varphi^{(n)} \quad \text{nach Definition von } \varphi^{(n+1)} \text{ und } \psi^{(n+1)} \\
 &= \underline{id}_{D_n} \circ d \circ \underline{id}_{D_n} \quad \text{nach Induktionsannahme} \\
 &= d
 \end{aligned}$$

2. Sei  $f \in D_{n+2}$ , dann gilt:

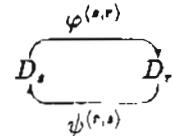
$$\begin{aligned}
 & \varphi^{(n+1)} \circ \psi^{(n+1)}(f) \\
 &= \varphi^{(n+1)}(\psi^{(n+1)}(f)) \\
 &= \varphi^{(n)} \circ \psi^{(n)} \circ f \circ \varphi^{(n)} \circ \psi^{(n)} \quad \text{nach Definition von } \varphi^{(n+1)} \text{ und } \psi^{(n+1)} \\
 &\subseteq \underline{id}_{D_{n+1}} \circ f \circ \underline{id}_{D_{n+1}} \quad \text{nach Induktionsannahme} \\
 &= f
 \end{aligned}$$

Q.E.D.

Durch Komposition der oben definierten Funktionen erhält man Retraktionspaare zu je zwei beliebigen cpo's  $D_s$  und  $D_r$ .

**Definition 3.32 (Retraktionspaare  $(\varphi^{(s,r)}, \psi^{(r,s)})$ )**

Seien  $s, r \in \mathbb{N}$  mit  $s \leq r$ . Das Retraktionspaar



ist induktiv gegeben durch:

1.  $\varphi^{(s,s)} = \underline{id}_{D_s}$ ,  
 $\psi^{(s,s)} = id_{D_s}$  und
2.  $\varphi^{(s+1,r)} = \varphi^{(r)} \circ \varphi^{(s,r)}$ ,  
 $\psi^{(r+1,s)} = \psi^{(s)} \circ \psi^{(r,s)}$ .

■

**Lemma 3.33** Für alle  $s, r \in \mathbb{N}$  mit  $s \leq r$  gilt:

$$(\varphi^{(s,r)}, \psi^{(r,s)})$$

ist ein Retraktionspaar.

**Beweis:** Die Stetigkeit beider Abbildungen folgt wieder aus Lemma 3.14, und die Retraktionseigenschaften folgen induktiv aus den Retraktionseigenschaften der Identitäten und der schon definierten Retraktionspaare.

Q.E.D.

Der direkte und inverse Limes einer Folge von semantischen Bereichen  $D_n$  ist der Scott-cpo  $D_\infty$ .

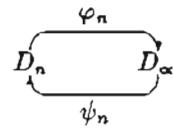
### Definition 3.34 ( $D_\infty$ )

Seien die semantischen Bereiche  $D_i$  ( $i \in \text{IN}$ ) wie oben konstruiert, dann ist der Scott-cpo  $D_\infty$  definiert durch

$$D_\infty := \{\langle d_\nu \mid \nu \in \text{IN} \rangle \mid d_\nu \in D_\nu \text{ und } \psi^{(\nu)}(d_{\nu+1}) = d_\nu\}$$

mit komponentenweiser cpo-Struktur.

Zu  $n \in \text{IN}$  ist das Paar



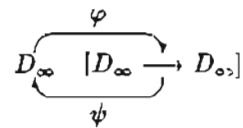
bestimmt durch:

$$\varphi_n(d)_\nu = \begin{cases} \psi^{(n,\nu)}(d) & \text{falls } n \geq \nu \\ \varphi^{(n,\nu)}(d) & \text{falls } n < \nu \end{cases}$$

und

$$\psi_n(\langle d_\nu \mid \nu \in \text{IN} \rangle) = d_n .$$

Das Funktionenpaar



ist definiert durch:

$$\varphi(\langle d_\nu \mid \nu \in \text{IN} \rangle) = \bigsqcup_{n \in \text{IN}} \varphi_n \circ d_{n+1} \circ \psi_n$$

und

$$\psi(f) = \bigsqcup_{n \in \text{IN}} \varphi_{n+1}(\psi_n \circ f \circ \varphi_n) .$$

■

**Lemma 3.35** Sei  $n \in \mathbb{N}$ .

Das Paar  $(\varphi_n, \psi_n)$  ist stetig und erfüllt die Retraktionseigenschaften.

**Beweis:** Die Stetigkeit folgt aus der Tatsache, daß die cpo-Struktur auf  $D_\infty$  komponentenweise erklärt ist. Die Retraktionseigenschaften ergeben sich aus denen von  $\varphi^{(n,\nu)}$  und  $\psi^{(n,\nu)}$  für alle  $\nu \in \mathbb{N}$ .

Q.E.D.

Das nächste Lemma zeigt, daß sich jedes Element aus dem Scott-cpo  $D_\infty$  als Limes seiner endlichen Approximationen gewinnen läßt.

**Lemma 3.36** Sei  $d = \langle d_\nu \mid \nu \in \mathbb{N} \rangle \in D_\infty$ .

Es gilt

$$d = \bigsqcup_{\nu \in \mathbb{N}} \varphi_\nu(d_\nu).$$

**Beweis:** Für beliebiges  $n \in \mathbb{N}$  gilt:

$$\begin{aligned} \psi_n(\bigsqcup_{\nu \in \mathbb{N}} \varphi_\nu(d_\nu)) &= \bigsqcup_{\nu \in \mathbb{N}} \psi_n(\varphi_\nu(d_\nu)) \quad \text{wegen der Stetigkeit von } \psi_n \\ &= d_n \quad \text{nach Definition von } D_\infty \\ &= \psi_n(d) \quad \text{nach Definition von } \psi_n \end{aligned}$$

Also ist  $d$  in allen Komponenten gleich  $\bigsqcup_{\nu \in \mathbb{N}} \varphi_\nu(d_\nu)$ .

Q.E.D.

Nun läßt sich zeigen, daß der semantische Bereich  $D_\infty$  eine Lösung der Bereichsgleichung

$$D = [D \longrightarrow D]$$

ist, und damit als Modell des reinen  $\lambda$ -Kalküls dient.

**Satz 3.37 (Scott)**

Der Bereich  $D_\infty$  ist homöomorph zu  $[D_\infty \longrightarrow D_\infty]$  mittels des Retraktionspaars  $(\varphi, \psi)$ , d.h.  $\varphi$  und  $\psi$  sind stetige Funktionen mit den Eigenschaften

$$1. \psi \circ \varphi = id_{D_\infty} \text{ und}$$

$$2. \varphi \circ \psi = id_{[D_\infty \longrightarrow D_\infty]}.$$

**Beweis:** Die Stetigkeit von  $\varphi$  und  $\psi$  folgt aus Lemma 3.14.

zu 1.: Sei  $d = \langle d_\nu \mid \nu \in \mathbb{N} \rangle \in D_\infty$ .

$$\begin{aligned}
 & \psi(\varphi(d)) \\
 &= \psi\left(\bigsqcup_{\nu \in \mathbb{N}} \varphi_\nu \circ d_{\nu+1} \circ \psi_\nu\right) && \text{nach Definition von } \varphi \\
 &= \bigsqcup_{n \in \mathbb{N}} \varphi_{n+1}\left(\psi_n \circ \left(\bigsqcup_{\nu \in \mathbb{N}} \varphi_\nu \circ d_{\nu+1} \circ \psi_\nu\right) \circ \varphi_n\right) && \text{nach Definition von } \psi \\
 &= \bigsqcup_{n \in \mathbb{N}} \varphi_{n+1}\left(\psi_n \circ \varphi_n \circ d_{n+1} \circ \psi_n \circ \varphi_n\right) && \text{wegen der Stetigkeit von } \\
 &&& \varphi_{n+1} \text{ und } \psi_n \text{ und der} \\
 &&& \text{Retraktionseigenschaften} \\
 &= \bigsqcup_{n \in \mathbb{N}} \varphi_{n+1}(d_{n+1}) && \text{wegen der Retraktions-} \\
 &&& \text{eigenschaften} \\
 &= \bigsqcup_{n \in \mathbb{N}} \varphi_n(d_n) && \text{da } \varphi_0(d_0) \sqsubseteq \varphi_1(d_1) \\
 &= d && \text{nach Lemma 3.36}
 \end{aligned}$$

zu 2.: Sei  $f \in [D_\infty \rightarrow D_\infty]$ .

$$\begin{aligned}
 & \varphi(\psi(f)) \\
 &= \varphi\left(\bigsqcup_{\nu \in \mathbb{N}} \varphi_{\nu+1}(\psi_\nu \circ f \circ \varphi_\nu)\right) && \text{nach Definition von } \psi \\
 &= \bigsqcup_{n \in \mathbb{N}} \varphi_n \circ \psi_{n+1}\left(\bigsqcup_{\nu \in \mathbb{N}} \varphi_{\nu+1}(\psi_\nu \circ f \circ \varphi_\nu)\right) \circ \psi_n && \text{nach Definition von } \varphi \\
 &= \bigsqcup_{n \in \mathbb{N}} \varphi_n \circ (\psi_n \circ f \circ \varphi_n) \circ \psi_n && \text{wegen der Stetigkeit von } \\
 &&& \varphi_n \text{ und } \psi_{n+1} \text{ und der} \\
 &&& \text{Retraktionseigenschaften} \\
 &= \bigsqcup_{n \in \mathbb{N}} (\varphi_n \circ \psi_n) \circ f \circ \bigsqcup_{n \in \mathbb{N}} (\varphi_n \circ \psi_n) \\
 &= \underline{id}_{D_\infty} \circ f \circ \underline{id}_{D_\infty} && \text{nach Lemma 3.36} \\
 &= f .
 \end{aligned}$$

Q.E.D.

Damit ist die gewünschte Konstruktion abgeschlossen. Es läßt sich weiter zeigen, daß der hier entwickelte semantische Bereich  $D_\infty$  neben den Unterbereichen  $D_n$  auch alle anderen endlich getypten Bereiche enthält, die sich aus den Basisbereichen durch die in Abschnitt 3.1 angegebenen Konstruktionen gewinnen lassen.

Im folgenden wird bei der Definition der semantischen Funktionen und anderer Elemente rekursiv definierter cpo's die Verwendung der Homöomorphismen implizit sein, d.h. z.B. daß man für Elemente  $d_1$  und  $d_2$  aus  $\bar{D}_\infty$  anstatt  $\varphi(d_1)(d_2)$  und  $\varphi(d_2)(d_1)$  einfach  $d_1(d_2)$  und  $d_2(d_1)$  schreiben kann.

## Kapitel 4

# Detaillierte Behandlung der denotationellen Semantik

In diesem Kapitel werden mit Hilfe der Theorie der semantischen Bereiche und des  $\lambda$ -Kalküls eine Reihe von semantischen Konzepten (spezielle Bereichskonstruktionen und darauf definierte Operationen) vorgestellt, die es erlauben, alle in imperativen Sprachen wie PASCAL, ALGOL 60 u.ä. vorkommenden Konstrukte in geeigneter Weise denotationell zu beschreiben.

Nach einer gewissen Anreicherung der Metasprache in Abschnitt 4.1 wird in Abschnitt 4.2 anhand der aus Kapitel 2 bekannten Beispielsprache WHILE gezeigt, wie sich die denotationelle Semantik mit Hilfe der in Kapitel 3 eingeführten Techniken durch Angabe weniger, leicht lesbarer Formeln definieren lässt. In Abschnitt 4.3 werden *Fortsetzungsfunktionen* eingeführt, mit denen die denotationelle Semantik des „Programmrestes“ adäquat definiert werden kann. Damit wird eine einfache Behandlung der Semantik von Sprunganweisungen und Fehlern ermöglicht.

In Abschnitt 4.4 wird eine neue, umfangreichere Sprache *PASCAL<sub>0</sub>* vorgestellt und mittels der Fortsetzungssemantik denotationell spezifiziert. Anschließend wird in Abschnitt 4.5 gezeigt, wie sich weitere aus der Programmierung bekannte Konzepte in diese Sprache einbetten und denotationell beschreiben lassen.

### 4.1 Spezielle Funktionen und Konventionen

In diesem Abschnitt werden einige Erweiterungen der Metasprache vorgestellt, die im getypten  $\lambda$ -Kalkül normalerweise nicht verwendet werden, aber für die Formulierung der denotationellen Semantik zweckmäßig sind.

### 4.1.1 Curry-Isomorphismen

Bei der Spezifikation der denotationellen Semantik einer Programmiersprache hat man es mit Funktionen mehrerer Veränderlicher zu tun. Schon bei der einfachen Sprache WHILE benötigt man zur Bestimmung des Wertes einer Anweisung  $C$  nicht nur  $C$  selbst, sondern auch noch einen Zustand  $z$ , in dem  $C$  ausgeführt werden soll. Man könnte nun meinen, daß daher die Semantikfunktion  $\mathcal{C}$  vom Typ

$$(COM \times ZUSTAND) \longrightarrow (ZUSTAND + \{\text{Fehler}\})$$

sei. Wie aber bereits in Kapitel 2 erläutert wurde, ist es zweckmäßig, die Funktion  $C$  vom Typ

$$COM \longrightarrow [ZUSTAND \longrightarrow (ZUSTAND + \{\text{Fehler}\})]$$

zu vereinbaren, um von der Semantik einer Anweisung als eigenständiges Objekt, nämlich einer Zustandstransformation, sprechen zu können. Diese Alternative der Typisierung beruht auf der in Abschnitt 2.4 erwähnten Isomorphie nach Curry zwischen  $(A \times B) \longrightarrow C$  und  $A \longrightarrow [B \longrightarrow C]$  für beliebige Bereiche  $A$ ,  $B$  und  $C$ . Diese Isomorphie soll nun in ihrer allgemeinen Form als  $\lambda$ -Ausdruck definiert werden.

#### Definition 4.1 (Curry-Isomorphismen)

Seien  $D, D_1, \dots, D_r, r \in IN$  semantische Bereiche, und sei  $f : D_1 \times \dots \times D_r \longrightarrow D$  eine stetige Funktion. Dann ist

$$\underline{\text{curry}}\ f : D_1 \longrightarrow [D_2 \longrightarrow \dots [D_r \longrightarrow D] \dots]$$

definiert durch

$$\underline{\text{curry}}\ f\ d_1\ d_2 \dots d_r = f(d_1, d_2, \dots, d_r),$$

d.h.:  $\underline{\text{curry}} :=$

$$\lambda f d_1 \dots d_r. f(d_1, \dots, d_r) : [D_1 \times \dots \times D_r \longrightarrow D] \longrightarrow [D_1 \longrightarrow \dots \longrightarrow [D_r \longrightarrow D] \dots].$$

■

**Beispiel 4.2** Sei  $\underline{\text{plus}} \in \cup \mathcal{K}$  mit

$$\underline{\text{plus}}(x, y) = \begin{cases} x + y & \text{falls } x, y \in IN \\ \perp & \text{sonst} \end{cases},$$

dann ist

$$\underline{\text{curry plus}}\ x\ y = \underline{\text{plus}}(x, y)$$

für alle  $x, y \in IN_{\perp}$ . Damit läßt sich jede Funktion über  $IN_{\perp}$ , die eine Konstante  $k \in IN$  auf ihr Argument addiert, durch den Ausdruck

$$\lambda x. \underline{\text{curry plus}}\ k\ x$$

darstellen. Dieser Ausdruck lässt sich unter Verwendung der  $\eta$ -Reduktion noch zu dem Ausdruck

curry plus k

vereinfachen.

□

### 4.1.2 Die bedingte Verzweigung

Neben den schon in Abschnitt 3.1 eingeführten Basisfunktionen enthält die Familie  $\mathcal{K}$  der Konstanten bei der denotationellen Methode auch noch spezielle Funktionen zur Formulierung von Alternativen, das *Konditional* (die *Verzweigung*) cond.

**Definition 4.3 (cond)**

Zu jedem semantischen Bereich  $D$  existiert cond in der Menge  $\mathcal{K}^{(D \times D) \rightarrow \text{BOOL} \rightarrow D}$  mit

$$\text{cond } \langle d_1, d_2 \rangle b = \begin{cases} d_1 & \text{falls } b = \text{wahr} \\ d_2 & \text{falls } b = \text{falsch} \\ \perp & \text{sonst} \end{cases} .$$

■

Man beachte, daß die Funktion cond zwar nicht strikt ist (z.B. gilt  $\text{cond } \langle d, \perp \rangle \text{ wahr} = d$ ), aber dennoch stetig.

**Konvention:** Zur Erhöhung der Lesbarkeit wird auch die Pfeilschreibweise nach McCarthy [83] verwendet. So steht

$$b \longrightarrow d_1, d_2$$

für cond  $\langle d_1, d_2 \rangle b$ ,

$$b_1 \longrightarrow d_1, b_2 \longrightarrow d_2, \dots, b_n \longrightarrow d_n, d_{n+1}$$

steht für  $b_1 \longrightarrow d_1, (b_2 \longrightarrow d_2, (\dots, (b_n \longrightarrow d_n, d_{n+1}) \dots))$  und

$$b_1 \longrightarrow d_1, b_2 \longrightarrow d_2, \dots, b_n \longrightarrow d_n$$

steht für  $b_1 \longrightarrow d_1, b_2 \longrightarrow d_2, \dots, b_n \longrightarrow d_n, \perp$ .

Die bedingte Verzweigung wird oft verwendet, wenn eine Funktion mit einem Argument aus einem Summenbereich  $D = D_1 + \dots + D_n$  definiert werden soll, und der  $i$ -te Zweig gerade den Fall behandelt, in dem das Argument Element aus dem Bereich  $D_i$  ist. Eine solche Definition hat dann folgende Form:

$$\begin{aligned} f = & \lambda x. \text{is}_{D_1} x \longrightarrow t_1, \\ & \vdots \\ & \text{is}_{D_n} x \longrightarrow t_n . \end{aligned}$$

Da in der Regel alle Glieder des Summenbereiches voneinander verschieden sind, kann man für solche Definitionen auch eine einfache Form der *Unifikation* benutzen, um bereits in den Testausdrücken  $\underline{is}_D; x$  die Struktur des jeweiligen Bereiches zu verdeutlichen und implizit seine Komponenten namentlich an den entsprechenden Ergebnisausdruck zu übergeben, wie das folgende Beispiel zeigt:

#### Beispiel 4.4

Zur Definition einer Funktion vom Typ

$$\text{IN}_\perp + (\mathbb{Z}_\perp \times \mathbb{Z}_\perp) + \{\text{Fehler}\}_\perp \longrightarrow \mathbb{Z}_\perp + \{\text{Fehler}\}_\perp$$

steht der Ausdruck

$$\begin{aligned} f &= \lambda x. \quad x = n \longrightarrow n, \\ &\quad x = \langle z_1, z_2 \rangle \longrightarrow z_1 + z_2, \\ &\quad x = \underline{\text{Fehler}} \longrightarrow \underline{\text{Fehler}} \end{aligned}$$

als Abkürzung für den Ausdruck

$$\begin{aligned} f &= \lambda x. \quad \underline{is}_{\text{IN}}, \quad x \longrightarrow \underline{out}_{\text{IN}}, \quad x, \\ &\quad \underline{is}_{\mathbb{Z}_\perp \times \mathbb{Z}_\perp}, \quad x \longrightarrow \pi_1(\underline{out}_{\mathbb{Z}_\perp \times \mathbb{Z}_\perp}, x) + \pi_2(\underline{out}_{\mathbb{Z}_\perp \times \mathbb{Z}_\perp}, x), \\ &\quad \underline{is}_{\{\text{Fehler}\}_\perp}, \quad x \longrightarrow \underline{out}_{\{\text{Fehler}\}_\perp}, \quad x. \end{aligned}$$

□

Formal lässt sich diese Notation wie folgt begründen:

Wenn  $d$  als Metavariable für Elemente aus  $D$  vereinbart ist, so steht eine Verzweigung der Form

$$x = d \longrightarrow t$$

als Abkürzung für den Ausdruck

$$\underline{is}_D \quad x \longrightarrow \hat{\delta}_{\underline{out}_D, x}^{d'}. t.$$

Wenn  $d_1, \dots, d_n$  als Metavariablen für Elemente aus  $D_1, \dots, D_n$  vereinbart sind, so steht eine Verzweigung der Form

$$x = \langle d_1, \dots, d_n \rangle \longrightarrow t$$

als Abkürzung für den Ausdruck

$$\underline{is}_{D_1 \times \dots \times D_n} \quad x \longrightarrow \$_{\pi_1(\underline{out}_{D_1 \times \dots \times D_n}, x)}^{d_1} \dots \$_{\pi_n(\underline{out}_{D_1 \times \dots \times D_n}, x)}^{d_n}.$$

Und schließlich, wenn  $d$  als Metavariable für Elemente aus  $D$  und  $d'$  als Metavariable für Elemente aus  $D'$  bzw. aus  $D^\omega$  vereinbart sind, so steht eine Verzweigung der Form

$$x = d.d' \longrightarrow t$$

als Abkürzung für den Ausdruck

$$\underline{is}_{D^*} x \longrightarrow \$_{hd(out_{D^*} x)}^d \$_{tl(out_{D^*} x)}^{d'} t ,$$

bzw. für den Ausdruck

$$\underline{is}_{D^*} x \longrightarrow \$_{hd(out_{D^*} x)}^d \$_{tl(out_{D^*} x)}^{d'} t .$$

### 4.1.3 Basisoperationen

Auf den Mengen, die den elementaren semantischen Bereichen zugrunde liegen, sind normalerweise einige *Grundoperationen* definiert. Z.B. die arithmetischen Operationen  $+, *, \dots$  auf  $\mathbb{Z}$ , die logischen Verknüpfungen  $\wedge, \vee, \dots$  auf der Menge der Wahrheitswerte  $\{\text{wahr}, \text{falsch}\}$  oder die Konkatenation von Worten über einem Alphabet über der entsprechenden Wortmenge. Diese Funktionen werden über den semantischen Bereichen  $\mathbb{Z}_\perp, \overline{BOOL}, ID$  usw. strikt erweitert und in der Bezeichnung nicht unterschieden.

#### Beispiel 4.5

$$4 + \perp = \perp$$

$$b_1 \vee b_2 = \begin{cases} b_1 \vee b_2 & \text{falls } b_1, b_2 \in \{\text{wahr}, \text{falsch}\} \\ \perp & \text{sonst} \end{cases}$$

□

### 4.1.4 Rekursion

Zur Formulierung von Rekursion in der Metasprache gibt es prinzipiell zwei Möglichkeiten. Entweder man verwendet explizit die Fixpunktoperatoren

$$\underline{fix} \in \mathcal{K}^{[D \rightarrow D] \rightarrow D}, d \in \mathcal{D}$$

(vgl. Satz 3.7 und Definition 3.17), oder man vereinbart die Rekursion implizit unter Verwendung der Gleichungsschreibweise. In der denotationellen Semantik bevorzugt man letzteres, daher soll hier genauer darauf eingegangen werden.

#### Beispiel 4.6 (Fakultätsfunktion)

Die Fakultätsfunktion ist über den natürlichen Zahlen durch folgende Gleichung definiert:

$$x! = \begin{cases} 1 & \text{falls } x = 0 \\ x * (x - 1)! & \text{sonst} \end{cases} .$$

□

Wie in Abschnitt 3.1 erwähnt wurde, soll der minimale Fixpunkt einer solchen Gleichung als Lösung vereinbart werden. In der Notation des  $\lambda$ -Kalküls lässt sich diese Aussage für eine gegebene Gleichung präzisieren. Betrachtet man obige Gleichung für die Fakultätsfunktion, so lässt sich ihr in natürlicher Weise folgende stetige Transformation zuordnen:

$$\lambda F. \lambda x. x = 0 \longrightarrow 1, x * F(x - 1) : [IN_{\perp} \longrightarrow IN_{\perp}] \longrightarrow [IN_{\perp} \longrightarrow IN_{\perp}] .$$

Die Fakultätsfunktion kann nun eindeutig als minimaler Fixpunkt dieses Funktions definiert werden:

$$\underline{fac} := \underline{fix} \lambda F. \lambda x. x = 0 \longrightarrow 1, x * F(x - 1) : [IN_{\perp} \longrightarrow IN_{\perp}] .$$

Da diese Schreibweise weniger gut lesbar ist, vereinbart man die Gleichung

$$\begin{aligned} \underline{fac} &= \lambda x. x = 0 \longrightarrow 1, x * \underline{fac}(x - 1) \quad \text{oder} \\ \underline{fac} x &= x = 0 \longrightarrow 1, x * \underline{fac}(x - 1) \end{aligned}$$

als Abkürzung für obige Definition. Damit liegt unsere Metasprache sehr nahe an der üblichen mathematischen Notation (vgl. 4.6).

**Allgemein:** Ein Gleichungssystem der Form

$$\begin{aligned} F_1 &= \tau_1 \\ F_2 &= \tau_2 \\ &\vdots \\ F_r &= \tau_r \end{aligned}$$

mit  $\tau_v \in \mathcal{A}_{\lambda}[\mathcal{D}, \mathcal{X} \cup \{F_1^{D_1}, \dots, F_r^{D_r}\}, \mathcal{K}]$  für  $1 \leq v \leq r$ , definiert

$$\langle F_1, \dots, F_r \rangle \in D_1 \times \dots \times D_r$$

durch

$$\langle F_1, \dots, F_r \rangle := \underline{fix} \lambda(F_1, \dots, F_r).(\tau_1, \dots, \tau_r) .$$

Im allgemeinen unterscheidet man in der Schreibweise nicht mehr zwischen den Funktionsvariablen  $F_v$  und den durch die entsprechenden Gleichungen definierten Funktionen  $F_v$ .

**Konvention:** Äußere Abstraktionen auf der rechten Seite einer Gleichung können als formale Parameter auf die linke Seite geschrieben werden, d.h. eine Gleichung der Form

$$F x_1 \dots x_s = \tau$$

steht als Abkürzung für die Gleichung

$$F = \lambda x_1 \dots x_s. \tau .$$

Am Beispiel der Fakultätsfunktion soll nun gezeigt werden, wie der minimale Fixpunkt einer Funktion approximiert (berechnet) werden kann. Sei

$$\tau := \lambda F. \lambda x. x = 0 \rightarrow 1, x * F(x - 1)$$

die der Gleichung für die Fakultätsfunktion zugeordnete Transformation. Aus Satz 3.7 folgt

$$\underline{fac} = \underline{fix} \tau = \bigsqcup_{n \in \mathbb{N}} \tau^n(\perp) \vdash \rightarrow \mathbb{N} \perp$$

Betrachtet man die ersten Glieder dieser Kette, so ergibt sich:

$$\begin{aligned}\tau^0(\perp) &= \perp \\ \tau^1(\perp) &= \lambda x. x = 0 \rightarrow 1, x * \perp(x - 1) \\ &= \lambda x. x = 0 \rightarrow 1, \perp \\ \tau^2(\perp) &= \lambda x. x = 0 \rightarrow 1, x * (\lambda x. x = 0 \rightarrow 1, \perp)(x - 1) \\ &= \lambda x. x = 0 \rightarrow 1, x * ((x - 1) = 0 \rightarrow 1, \perp) \\ &= \lambda x. x = 0 \rightarrow 1, x * (x = 1 \rightarrow 1, \perp) \\ &= \lambda x. (x = 0 \vee x = 1) \rightarrow 1, \perp \\ \tau^3(\perp) &= \lambda x. x = 0 \rightarrow 1, x * (\lambda x. (x = 0 \vee x = 1) \rightarrow 1, \perp)(x - 1) \\ &= \lambda x. x = 0 \rightarrow 1, x * ((x = 1 \vee x = 2) \rightarrow 1, \perp) \\ &= \lambda x. (x = 0 \vee x = 1) \rightarrow 1, x = 2 \rightarrow 2, \perp \\ &\vdots \\ \tau^n(\perp) &= \lambda x. x \leq n \rightarrow x!, \perp\end{aligned}$$

Man sieht insbesondere, daß für jedes Argument  $n \in \mathbb{N}$  zur Berechnung von  $\underline{fac} n$  die ersten  $n + 1$  Approximationen genügen. Aus der Ordnungsrelation für den Funktionenbereich ergibt sich nämlich: Wenn

$$\tau^i(\perp)(n) = m \neq \perp$$

gilt, so gilt auch  $\tau^k(\perp)(n) = m$  für alle  $k \geq i$ , und damit gilt auch

$$\underline{fac} n = m .$$

#### 4.1.5 Modifikation von Funktionen

So wie in Definition 3.19 Umgebungen an einer Stelle modifiziert wurden, sollen auch beliebige Funktionen an bestimmten Stellen abgeändert werden können.

**Definition 4.7** Sei  $f : D \rightarrow D'$  eine stetige Funktion über semantischen Bereichen  $D$  und  $D'$  und seien  $d$  und  $d'$  je ein Element aus  $D$  und  $D'$ . Die Modifikation

von  $f$  an der Stelle  $d$  mit neuem Wert  $d'$  wird mit  $f[d'/d]$  bezeichnet, d.h.:

$$\begin{aligned} f[d'/d] &\in [D \rightarrow D'] \text{ mit} \\ x &\mapsto \begin{cases} d' & \text{falls } x = d \\ f(x) & \text{sonst} \end{cases} . \end{aligned}$$

Für eine Änderung von  $f$  an mehreren Stellen steht der Ausdruck

$$f[d'_1 d'_2 \cdots d'_n / d_1 d_2 \cdots d_n]$$

als Abkürzung für den Ausdruck

$$f[d'_1/d_1] [d'_2/d_2] \cdots [d'_n/d_n] .$$

■

#### 4.1.6 Die verallgemeinerte Komposition

Die Komposition  $\circ$  von Funktionen lässt sich durch den  $\lambda$ -Ausdruck

$$\begin{aligned} g \circ f &= \lambda x. g(f(x)) \text{ bzw.} \\ \circ &= \lambda g f x. g(f(x)) \end{aligned}$$

formalisieren. Dieser Operator lässt sich nur auf Funktionen anwenden, deren Typen von der Form

$$f : D_1 \rightarrow D_2 \text{ und } g : D_2 \rightarrow D_3$$

sind. Bei der denotationellen Semantikspezifikation sollen jedoch häufig zwei Operationen hintereinander ausgeführt werden, die jeweils als Ergebnis eine Fehlermeldung liefern können und deren Ziel- bzw. Argumentbereich nur nach Anwendung der Curry-Isomorphie übereinstimmen, d.h.

$$f : D_1 \rightarrow (D_2 + \{\underline{\text{Fehler}}\}) \text{ und } g : D_2 \rightarrow (D_3 + \{\underline{\text{Fehler}}\})$$

oder

$$f : D_1 \rightarrow ((D_2 \times \cdots \times D_n) + \{\underline{\text{Fehler}}\}) \text{ und}$$

$$g : D_2 \rightarrow \cdots \rightarrow D_n \rightarrow (D_{n+1} + \{\underline{\text{Fehler}}\}) .$$

Um die erforderlichen Anpassungen nicht immer explizit durchführen zu müssen, definiert man den Operator  $\star$  zur Komposition solcher Funktionen. Dabei soll  $f \star g$  ähnlich wie  $g \circ f$  wirken mit zwei Abweichungen:

1. Wenn  $f x = \underline{\text{Fehler}}$  gilt, so soll auch  $(f \star g) x = \underline{\text{Fehler}}$  sein, und
2. wenn  $f x = \langle d_2, \dots, d_n \rangle \in (D_2 \times \cdots \times D_n)$  gilt und  $g$  vom Typ  $D_2 \rightarrow \cdots \rightarrow D_n \rightarrow (D_{n+1} + \{\underline{\text{Fehler}}\})$  ist, so soll das Ergebnis von  $(f \star g) x$  gleich der aufeinanderfolgenden Anwendung von  $g$  auf die Elemente  $d_2$  bis  $d_n$  sein.

Die formale Definition lautet wie folgt:

**Definition 4.8 (Der Operator  $\star$ )**

1. Sei  $f : D_1 \longrightarrow (D_2 + \{\underline{\text{Fehler}}\})_1$  und  $g : D_2 \longrightarrow (D_3 + \{\underline{\text{Fehler}}\})_1$ , dann ist

$$\begin{aligned} f \star g &= \lambda x. \quad f x = \underline{\text{Fehler}} \longrightarrow \underline{\text{Fehler}}, \\ f x = d &\longrightarrow g d . \end{aligned}$$

2. Sei  $f : D_1 \longrightarrow ((D_2 \times \dots \times D_n) + \{\underline{\text{Fehler}}\})_1$  und

$$g : D_2 \longrightarrow \dots \longrightarrow D_n \longrightarrow (D_{n+1} + \{\underline{\text{Fehler}}\})_1, \text{ dann ist}$$

$$\begin{aligned} f \star g &= \lambda x. \quad f x = \underline{\text{Fehler}} \longrightarrow \underline{\text{Fehler}}, \\ f x = \langle d_2, \dots, d_n \rangle &\longrightarrow g d_2 \dots d_n . \end{aligned}$$

■

**Bemerkung:** Man prüft leicht nach, daß die Operation  $\star$  ebenso wie die Komposition  $\circ$  assoziativ ist, so daß man bei einer mehrfachen Anwendung auf eine Klammerung verzichten kann.

## 4.2 Denotationelle Semantik der Sprache WHILE unter Verwendung der neuen Notationen

Der Gewinn, den man aus der Bereitstellung der im vorangegangenen Abschnitt eingeführten Techniken und Schreibweisen erhält, lässt sich am besten anhand des bekannten Beispiels der Sprache WHILE verdeutlichen. Daher wird die denotationelle Semantik von WHILE an dieser Stelle noch einmal spezifiziert (vgl. Abschnitte 2.1 und 2.4).

### Syntaktische Bereiche mit dazugehörigen Metavariablen

$Z \in ZAHL$ ,  $W \in BOOL$ ,  $K \in KON$ ,  $I \in ID$ ,  $OP \in OP$ ,  $BOP \in BOP$ ,  $T \in TERM$ ,  $B \in BT$ ,  $C \in COM$  und  $P \in PROG$ .

### Syntaktische Klauseln

$$\begin{aligned} T &::= Z \mid I \mid T_1 \underline{OP} T_2 \mid \text{read} \\ B &::= W \mid \text{not } B \mid T_1 \underline{BOP} T_2 \mid \text{read} \\ C &::= \text{skip} \mid I := T \mid C_1, C_2 \mid \text{if } B \text{ then } C_1 \text{ else } C_2 \mid \text{while } B \text{ do } C \mid \\ &\quad \text{output } T \mid \text{output } B \\ P &::= C \end{aligned}$$

Semantische Bereiche

$$\begin{aligned}
 ZUSTAND &= SPEICHER \times EINGABE \times AUSGABE \\
 SPEICHER &= ID \longrightarrow (\mathbb{Z}_\perp + \{\underline{frei}\}_\perp) \\
 EINGABE &= \overline{KON}^* \\
 AUSGABE &= \overline{KON}^*
 \end{aligned}$$

Semantikfunktionen

$$\begin{aligned}
 T : TERM &\longrightarrow ZUSTAND \longrightarrow ((\mathbb{Z}_\perp \times ZUSTAND) + \{\underline{Fehler}\}_\perp) \\
 B : BT &\longrightarrow ZUSTAND \longrightarrow ((\overline{BOOL} \times ZUSTAND) + \{\underline{Fehler}\}_\perp) \\
 C : COM &\longrightarrow ZUSTAND \longrightarrow (ZUSTAND + \{\underline{Fehler}\}_\perp) \\
 P : PROG &\longrightarrow \overline{KON}^* \longrightarrow (\overline{KON}^* + \{\underline{Fehler}\}_\perp)
 \end{aligned}$$

Semantische Klauseln

## 1. Semantik der Terme

- (a)  $T[n]z = \langle n, z \rangle$  für alle  $n \in ZAHL$
- (b)  $T[I](s, e, a) = sI = \underline{frei} \longrightarrow \underline{Fehler}. \langle sI, \langle s, e, a \rangle \rangle$
- (c)  $T[T_1 + T_2] = T[T_1] * \lambda n_1. T[T_2] * \lambda n_2 z. \langle n_1 + n_2, z \rangle$
- (d) analog für alle anderen arithmetischen Operationen
- (e)  $T[\text{read}] \langle s, e, a \rangle = \underline{null} e \longrightarrow \underline{Fehler}. \text{is } \underline{x}. \langle \underline{hd} e \rangle \longrightarrow \langle \underline{hd} e, \langle s, \underline{tl} e, a \rangle \rangle. \underline{Fehler}$

## 2. Semantik der booleschen Terme

- (a)  $B[\text{true}]z = \langle wahr, z \rangle$
- (b)  $B[\text{false}]z = \langle falsch, z \rangle$
- (c)  $B[\text{not } B] = B[B] * \lambda bz. \langle \neg b, z \rangle$
- (d)  $B[T_1 = T_2] = T[T_1] * \lambda n_1. T[T_2] * \lambda n_2 z. \langle n_1 = n_2, z \rangle$
- (e) analog für alle anderen booleschen Operationen

$$(f) \quad \mathcal{B}[\text{read }](s, e, a) = \underline{\text{null}} \ e \longrightarrow \underline{\text{Fehler}},$$

$$\underline{\text{is\_BOOLE}}(\underline{\text{hd}} \ e) \longrightarrow \langle \underline{\text{hd}} \ e, \langle s, \underline{\text{tl}} \ e, a \rangle \rangle, \underline{\text{Fehler}}$$

### 3. Semantik der Anweisungen

$$(a) \quad \mathcal{C}[\text{skip }]z = z$$

$$(b) \quad \mathcal{C}[I := T] = \mathcal{T}[T] * \lambda n(s, e, a). \langle s[n/I], e, a \rangle$$

$$(c) \quad \mathcal{C}[C_1; C_2] = \mathcal{C}[C_1] * \mathcal{C}[C_2]$$

$$(d) \quad \mathcal{C}[\text{if } B \text{ then } C_1 \text{ else } C_2] = \mathcal{B}[B] * \underline{\text{cond}}(\mathcal{C}[C_1], \mathcal{C}[C_2])$$

$$(e) \quad \mathcal{C}[\text{while } B \text{ do } C] = \mathcal{B}[B] * \underline{\text{cond}}(\mathcal{C}[C] * \mathcal{C}[\text{while } B \text{ do } C], \lambda z.z)$$

$$(f) \quad \mathcal{C}[\text{output } T] = \mathcal{T}[T] * \lambda n(s, e, a). \langle s, e, a.n \rangle$$

$$(g) \quad \mathcal{C}[\text{output } B] = \mathcal{B}[B] * \lambda b(s, e, a). \langle s, e, a.b \rangle$$

### 4. Semantik der Programme

$$\mathcal{P}[C]e = (\mathcal{C}[C] * \pi_3)(s_0, e, \varepsilon),$$

wobei  $s_0 \in \text{SPEICHER}$  gegeben ist durch  $s_0 I = \underline{\text{frei}}$  für alle  $I \in ID$ .

Die Äquivalenz dieser Beschreibung zu derjenigen aus Abschnitt 2.4 lässt sich unmittelbar aus den Definitionen von cond und  $*$  sowie den Gesetzen des  $\lambda$ -Kalküls herleiten. Mit der jetzigen Definition wurde eine erhebliche Abkürzung erreicht, so daß der geübte Leser die Bedeutung der einzelnen Konstrukte der Sprache WHILE auf einen Blick lesen und verstehen kann. Zur Illustration der Wirkungsweise der neu eingeführten Operatoren, wird die Semantik eines kurzen Beispielprogramms, angewendet auf eine konkrete Eingabe, berechnet.

Gegeben sei folgendes WHILE-Programm:

$$P \left\{ \begin{array}{l} x := \text{read}; \\ \text{while } x > 0 \text{ do} \\ \quad C \left\{ \begin{array}{l} \text{output } x + x; \\ x := x - 1 \end{array} \right. \end{array} \right.$$

Die Semantik von  $P$ , angewendet auf die Eingabedatei  $\langle 1 \rangle$ , lässt sich nun genau bestimmen. Dazu werden einige Nebenrechnungen ( $N1 - N7$ ) benötigt, die im

Anschluß aufgeführt sind. Zur Abkürzung der Zustandsbezeichnungen sei

$$\begin{aligned} z_0 &= \langle s_0, \langle 1 \rangle, \epsilon \rangle, \\ z_1 &= \langle s_0 [1/x], \epsilon, \epsilon \rangle \text{ und} \\ z_2 &= \langle s_0 [0/x], \epsilon, \langle 2 \rangle \rangle. \end{aligned}$$

Nun gilt:  $\mathcal{C}[P] z_0$

$$\begin{aligned} &= (\mathcal{C}[x := \text{read}] * \mathcal{C}[\text{while } x > 0 \text{ do } C]) z_0 && \text{nach 3c} \\ &= \mathcal{C}[\text{while } x > 0 \text{ do } C] z_1 && \text{nach 4.8.1, N1} \\ &= (\mathcal{B}[x > 0] * \underline{\text{cond}}(\mathcal{C}[C] * \mathcal{C}[\text{while } x > 0 \text{ do } C], \lambda z.z)) z_1 && \text{nach 3e} \\ &= \underline{\text{cond}}(\mathcal{C}[C] * \mathcal{C}[\text{while } x > 0 \text{ do } C], \lambda z.z) \text{ wahr } z_1 && \text{nach 4.8.2, N3} \\ &= (\mathcal{C}[C] * \mathcal{C}[\text{while } x > 0 \text{ do } C]) z_1 && \text{nach 4.3} \\ &= \mathcal{C}[\text{while } x > 0 \text{ do } C] z_1 && \text{nach 4.8.1, N5} \\ &= (\mathcal{B}[x > 0] * \underline{\text{cond}}(\mathcal{C}[C] * \mathcal{C}[\text{while } x > 0 \text{ do } C], \lambda z.z)) z_2 && \text{nach 3e} \\ &= \underline{\text{cond}}(\mathcal{C}[C] * \mathcal{C}[\text{while } x > 0 \text{ do } C], \lambda z.z) \text{ falsch } z_2 && \text{analog zu N3} \\ &= (\lambda z.z) z_2 && \text{nach 4.3} \\ &= z_2 && \beta\text{-Red.} \end{aligned}$$

$$\begin{aligned} \text{Also folgt: } \mathcal{P}[P](1) &= (\mathcal{C}[P] * \pi_3) z_0 && \text{nach 4} \\ &= \pi_3 z_2 && \text{nach 4.8.1} \\ &= \langle 2 \rangle \end{aligned}$$

**Nebenrechnungen:** Sei  $z_3 = \langle s_0 [1/x], \epsilon, \langle 2 \rangle \rangle$ .

N1)  $\mathcal{C}[x := \text{read}] z_0$

$$\begin{aligned} &= (\mathcal{T}[\text{read}] * \lambda n(s, e, a). \langle s[n/x], e, a \rangle) z_0 && \text{nach 3b} \\ &= (\lambda n(s, e, a). \langle s[n/x], e, a \rangle) 1 z_0 && \text{nach 4.8.2, N2} \\ &= \langle s_0 [1/x], \epsilon, \epsilon \rangle && \beta\text{-Red.} \\ &= z_1 \end{aligned}$$

N2)  $\mathcal{T}[\text{read}] z_0 = \langle 1, \langle s_0, \epsilon, \epsilon \rangle \rangle$  nach 1e

N3)  $\mathcal{B}[x > 0] z_1$

$$\begin{aligned} &= (\mathcal{T}[x] * \lambda n_1. \mathcal{T}[0] * \lambda n_2 z. \langle n_1 > n_2, z \rangle) z_1 && \text{nach 2e} \\ &= (\lambda n_1. \mathcal{T}[0] * \lambda n_2 z. \langle n_1 > n_2, z \rangle) 1 z_1 && \text{nach 4.8.2, N4} \\ &= (\mathcal{T}[0] * \lambda n_2 z. \langle 1 > n_2, z \rangle) z_1 && \text{nach 4.8.2, 1a} \\ &= (\lambda n_2 z. \langle 1 > n_2, z \rangle) 0 z_1 && \beta\text{-Red.} \\ &= \langle 1 > 0, z_1 \rangle && \beta\text{-Red.} \\ &= \langle \text{wahr}, z_1 \rangle && \delta\text{-Red.} \end{aligned}$$

N4)  $T[x] z_1 = \langle 1, z_1 \rangle$  nach 1b

N5)  $C[C] z_1$

$$\begin{aligned}
 &= (\mathcal{C}[\text{output } x + x] * \mathcal{C}[x := x - 1]) z_1 \quad \text{nach 3c} \\
 &= \mathcal{C}[x := x - 1] z_3 \quad \text{nach 4.8.1, N6} \\
 &= (T[x - 1] * \lambda n(s, e, a).(s[n/x], e, a)) z_3 \quad \text{nach 3b} \\
 &= (\lambda n(s, e, a).(s[n/x], e, a)) 0 z_3 \quad \text{nach 4.8.2, N7} \\
 &= \langle s_0[0/x], \epsilon, \langle 2 \rangle \rangle \quad \delta\text{-Red.} \\
 &= z_2
 \end{aligned}$$

N6)  $C[\text{output } x + x] z_1$

$$\begin{aligned}
 &= (T[x + x] * \lambda n(s, e, a).(s, e, a.n)) z_1 \quad \text{nach 3f} \\
 &= (\lambda n(s, e, a).(s, e, a.n)) 2 z_1 \quad \text{nach 1c, 1b, 4.8.2} \\
 &= \langle s_0[1/x], \epsilon, \langle 2 \rangle \rangle \quad \beta\text{-Red.} \\
 &= z_3
 \end{aligned}$$

N7)  $T[x - 1] z_3$

$$\begin{aligned}
 &= (T[x] * \lambda n_1. T[1] * \lambda n_2 z.(n_1 - n_2, z)) z_3 \quad \text{nach 1d} \\
 &= (\lambda n_1. T[1] * \lambda n_2 z.(n_1 - n_2, z)) 1 z_3 \quad \text{nach 4.8.2, 1b} \\
 &= (T[1] * \lambda n_2 z.(1 - n_2, z)) z_3 \quad \beta\text{-Red.} \\
 &= (\lambda n_2 z.(1 - n_2, z)) 1 z_3 \quad \text{nach 4.8.2, 1a} \\
 &= \langle 0, z_3 \rangle \quad \beta\text{-Red. und } \delta\text{-Red.}
 \end{aligned}$$

### 4.3 Entwicklung der Standardsemantik unter besonderer Berücksichtigung der Fortsetzungssemantik

Mit den bisher vorgestellten Techniken lassen sich einige bekannte Sprachkonstrukte gar nicht oder nur schlecht beschreiben. Dies sind grob charakterisiert *Sprunganweisungen* und *gemeinsame Speicherplätze*. Zur Behandlung von Sprüngen, Fehlern und anderen Konstrukten, die eine Abweichung von der sequentiellen Programmausführung erzwingen, dient das Konzept der *Fortsetzungen (continuations)*. Gemeinsame Speicherplätze (*sharing, aliasing*) werden semantisch durch eine Aufspaltung der Bindung von Variablen an Werte beschrieben: Anstelle des Bereiches

$$SPEICHER = ID \longrightarrow (\mathbb{Z}_1 + \{frei\}_1)$$

verwendet man die beiden Bereiche  $ENV$  (für ‘environment’) und  $STORE$ :

$$\begin{aligned} ENV &= ID \longrightarrow (LOC + \{\underline{ungebunden}\}_\perp) \text{ und} \\ STORE &= LOC \longrightarrow (Z_\perp + \{\underline{frei}\}_\perp). \end{aligned}$$

I.a. bezeichnet man eine denotationelle Semantikfunktion, die sowohl mit Fortsetzungen arbeitet als auch das Umgebungs-/Speicherkonzept verwendet, als *Standardsemantik*.

Die nächsten Unterabschnitte führen formal in die Techniken der Standardsemantik ein.

#### 4.3.1 Fortsetzungen

Der  $\star$ -Operator dient bereits dazu, bei Auftreten eines Fehlers eine Sequenz von Funktionsanwendungen abzubrechen. In ähnlicher Weise müßte man für Sprunganweisungen auch vorgehen. Die normale Ausführung von  $C_1; C_2$  im Zustand  $z$  läßt sich ja schreiben als

$$\mathcal{C}[C_1; C_2]z = \mathcal{C}[C_2](\mathcal{C}[C_1]z)$$

Dies bedeutet, daß in der Semantik von  $C_2$  die Entscheidung gefällt wird, was mit dem Ergebnis von  $C_1$  zu tun ist. Eine bessere Möglichkeit wäre, wenn bei der Spezifikation der Semantik von  $C_1$  bereits die Entscheidung getroffen werden könnte, ob mit der Ausführung von  $C_2$  fortgesfahren oder eine irreguläre Fortsetzung des Programms durchgeführt werden soll. Dies wird dann möglich, wenn der Semantikfunktion von  $C_1$  nicht nur der aktuelle Zustand  $z$  sondern auch noch eine *Fortsetzung*  $c$ , die die Semantik des hinter  $C_1$  stehenden Programmrestes darstellt, als Argument vorliegt. Die Fortsetzung  $c$  ist also ein funktionales Objekt und beschreibt diejenige Zustandstransformation, die durch Ausführung aller auf  $C_1$  folgenden Anweisungen bestimmt ist. Sei z.B.  $P$  ein Programm, bestehend aus der Anweisungsfolge  $C_1; C_2; C_3$ , dann würde man der Fortsetzungssemantik von  $C_3$  die Identitätsfunktion  $\lambda z.z$  als Fortsetzung übergeben, derjenigen von  $C_2$  die Fortsetzung, die durch Ausführung von  $C_3$  bestimmt ist, und schließlich erhielte die Semantikfunktion zu  $C_1$  die Zustandstransformation  $\mathcal{C}[C_2; C_3]\lambda z.z$  als Fortsetzung.

Präziser formuliert, beschreibt man nun die Semantik einer Sequenz  $C_1; C_2$  bzgl. der Fortsetzung  $c$  und des Zustandes  $z$  durch die Formel

$$\mathcal{C}[C_1; C_2]cz = \mathcal{C}[C_1](\mathcal{C}[C_2]c)z.$$

Nun ist es möglich, daß der Zustand  $z'$ , der aus  $z$  durch Ausführung von  $C_1$  entsteht, je nach Bedeutung von  $C_1$  an die Fortsetzung  $\mathcal{C}[C_2]c$  oder an irgendeine andere Zustandstransformation übergeben wird.

Betrachtet man eine Sprache, in der Anweisungen der Form **goto L** erlaubt sind, und vereinbart, daß  $c_L$  gerade diejenige Zustandstransformation bezeichnet, die

einer Programmausführung ab der Marke  $L$  entspricht, dann kann man jetzt sehr einfach die Semantik von Sprunganweisungen erklären:

$$\mathcal{C}[\text{goto } L]cz = c_L z .$$

Man sieht an dieser Definition, daß die Semantik einer solchen Sprunganweisung völlig unabhängig von der aktuellen Fortsetzung  $c$  ist.

Wadsworth und Morris entwickelten unabhängig voneinander das Konzept der Fortsetzungssemantik. Eine erste Präsentation dieser Techniken findet man bei Strachey und Wadsworth [115]. De Bruin gibt eine formale Einführung in das Konzept der Fortsetzungssemantik und setzt es bzgl. einer einfachen Beispielsprache mit **goto's** in Beziehung zur operationellen und axiomatischen Semantik [19].

Natürlich kann auch bei der Berechnung von Termen ein irregulärer Kontrollfluß hervorgerufen werden, z.B. bei Auftreten eines Fehlers (etwa bei Ausführung von **read** in einem Zustand mit leerer Eingabe) oder bei Aufruf einer Funktionsprozedur. Also muß auch den Semantikfunktionen  $T$  und  $B$  ein Fortsetzungsparameter  $k$  bzw.  $g$  übergeben werden. Diese „Fortsetzung“ eines Terms  $T$  ist nun keine reine Zustandstransformation, sondern eine Funktion, die zu dem aus  $T$  errechneten Wert  $n$  eine Zustandstransformation bestimmt.

Bei regulärer Berechnung kann man also die Fortsetzungssemantik von Ausdrücken durch folgende Formeln beschreiben:

$$\begin{aligned} T[T]kz &= knz \\ T[B]gz &= gbz , \end{aligned}$$

wobei  $n$  den Wert von  $T$  bzgl.  $z$  bezeichnet und ebenso  $b$  den Wert von  $B$ .

Um den Umgang mit der Fortsetzungssemantik zu illustrieren, soll die Semantik der Sprache WHILE noch ein letztes Mal spezifiziert werden, obwohl in WHILE keine Sprunganweisungen vorhanden sind und daher kein verkürzender Effekt eintritt.

#### 4.3.2 Fortsetzungssemantik der Sprache WHILE

##### Syntaktische Bereiche und Klauseln

wie in Abschnitt 4.2.

Semantische Bereiche

$$\begin{aligned}
 ZUSTAND &= SPEICHER \times EINGABE \times AUSGABE \\
 SPEICHER &= ID \longrightarrow (\mathbb{Z}_1 + \{\underline{frei}\}_1) \\
 EINGABE &= \overline{KON^*} \\
 AUSGABE &= \overline{KON^*} \\
 FORTSETZUNG &= ZUSTAND \longrightarrow (ZUSTAND + \{\underline{Fehler}\}_1) \\
 TERMFORT &= \mathbb{Z}_1 \longrightarrow FORTSETZUNG \\
 BOOLFORT &= \overline{BOOL} \longrightarrow FORTSETZUNG
 \end{aligned}$$

Semantikfunktionen

$$\begin{aligned}
 T : TERM &\longrightarrow TERMFORT \longrightarrow FORTSETZUNG \\
 B : BT &\longrightarrow BOOLFORT \longrightarrow FORTSETZUNG \\
 C : COM &\longrightarrow FORTSETZUNG \longrightarrow FORTSETZUNG \\
 P : PROG &\longrightarrow \overline{KON^*} \longrightarrow (\overline{KON^*} + \{\underline{Fehler}\}_1)
 \end{aligned}$$

Semantische Klauseln

## 1. Semantik der Terme

- (a)  $T[n] k = k n$  für alle  $n \in ZAHL$
- (b)  $T[I] k \langle s, e, a \rangle = sI = \underline{frei} \longrightarrow \underline{Fehler}, k(sI) \langle s, e, a \rangle$
- (c)  $T[T_1 + T_2] k = T[T_1] \lambda n_1. T[T_2] \lambda n_2. k(n_1 + n_2)$
- (d) analog für alle anderen arithmetischen Operationen
- (e)  $T[\text{read}] k \langle s, e, a \rangle = \underline{null} e \longrightarrow \underline{Fehler},$   
 $\quad \quad \quad \quad \quad is_{\mathbb{Z}_1}(hd e) \longrightarrow k(hd e) \langle s, tl e, a \rangle, \underline{Fehler}$

## 2. Semantik der booleschen Terme

- (a)  $B[\text{true}] g = g wahr$
- (b)  $B[\text{false}] g = g falsch$
- (c)  $B[\text{not } B] g = B[B] \lambda b. g(\neg b)$

- (d)  $\mathcal{B} [T_1 = T_2] g = \mathcal{T} [T_1] \lambda n_1. \mathcal{T} [T_2] \lambda n_2. g(n_1 = n_2)$
- (e) analog für alle anderen booleschen Operationen
- (f)  $\mathcal{B} [\text{read}] g(s, e, a) - \underline{\text{null}} e \longrightarrow \underline{\text{Fehler}}$   
 $is_{BOOL}(\underline{hd} e) \longrightarrow g(\underline{hd} e)(s, \underline{tl} e, a), \underline{\text{Fehler}}$

### 3. Semantik der Anweisungen

- (a)  $\mathcal{C} [\text{skip}] c = c$
- (b)  $\mathcal{C} [I := T] c = \mathcal{T} [T] \lambda n(s, e, a). c(s[n/I], e, a)$
- (c)  $\mathcal{C} [C_1; C_2] = \mathcal{C} [C_1] \circ \mathcal{C} [C_2]$
- (d)  $\mathcal{C} [\text{if } B \text{ then } C_1 \text{ else } C_2] = \mathcal{B} [B] \underline{\text{cond}}(\mathcal{C} [C_1], \mathcal{C} [C_2])$
- (e)  $\mathcal{C} [\text{while } B \text{ do } C] = \mathcal{B} [B] \underline{\text{cond}}(\mathcal{C} [C] \circ \mathcal{C} [\text{while } B \text{ do } C], \underline{id})$
- (f)  $\mathcal{C} [\text{output } T] c = \mathcal{T} [T] \lambda n(s, e, a). c(s, e, a.n)$
- (g)  $\mathcal{C} [\text{output } B] c = \mathcal{B} [B] \lambda b(s, e, a). c(s, e, a.b)$

### 4. Semantik der Programme

$$\mathcal{P} [C] e = (\mathcal{C} [C] (\lambda z.z) \star \pi_3) (s_0, e, \varepsilon),$$

wobei  $s_0 \in SPEICHER$  gegeben ist durch  $s_0 I = \underline{frei}$  für alle  $I \in ID$ .

#### Bemerkungen zur Fortsetzungssemantik von WHILE

An den semantischen Klauseln der Fortsetzungssemantik von WHILE erkennt man schon, wie mit Fortsetzungen gearbeitet wird. Die Ausdrücke, die im  $\lambda$ -Kalkül die Fortsetzungen beschreiben, werden so aufgebaut, daß alle vorher zu berechnenden Parameter namentlich übergeben werden können. Besonders deutlich wird dies an der Klausel 3b: Die Berechnung des Terms  $T$  liefert den Wert  $n$  und den neuen Zustand  $(s, e, a)$  als Parameter an die Ausdrucksfortsetzung von  $T$ . Diese übergibt nun denjenigen Zustand an die Anweisungsfortsetzung  $c$ , der sich aus der entsprechenden Wertzuweisung ergibt.

**Beispiel 4.9**

$$\begin{aligned}
 & \mathcal{C}[I := \text{read}] c \langle S, 4.E, A \rangle \\
 &= T[\text{read}] (\lambda n(s, e, a). c(s[n/I], e, a)) \langle S, 4.E, A \rangle \quad \text{nach 3b} \\
 &= (\lambda n(s, e, a). c(s[n/I], e, a)) 4 \langle S, E, A \rangle \quad \text{nach 1e} \\
 &= c \langle S[4/I], E, A \rangle \quad \beta\text{-Reduktion}
 \end{aligned}$$

□

Interessant ist auch, daß man nun bei der Semantikspezifikation der Terme und Anweisungen ganz auf den  $\star$ -Operator verzichten kann. Mit der Hintereinanderausführung  $\circ$  werden jetzt die Fortsetzungen eines Programms von hinten nach vorne transformiert. Wenn bei der Berechnung einer Anweisung ein Fehler auftritt, so wird eine Fehlermeldung erzeugt und der Rest des Programms (die aktuelle Fortsetzung) ignoriert.

**Beispiel 4.10**

$$\begin{aligned}
 & \mathcal{C}[I := \text{read} ; \text{output } 7] (\lambda z.z) \langle S, \langle \rangle, A \rangle \\
 &= \mathcal{C}[I := \text{read}] (\mathcal{C}[\text{output } 7] \lambda z.z) \langle S, \langle \rangle, A \rangle \quad \text{nach 3c} \\
 &= T[\text{read}] (\lambda n(s, e, a). \mathcal{C}[\text{output } 7] (\lambda z.z) (s[n/I], e, a)) \langle S, \langle \rangle, A \rangle \quad \text{nach 3b} \\
 &= \underline{\text{Fehler}} \quad \text{nach 1e}
 \end{aligned}$$

□

**4.3.3 Typüberprüfung**

Bei der Spezifikation der denotationellen Semantik lassen sich die semantischen Klauseln stets einer ersten Prüfung unterziehen, indem man die verwendeten Ausdrücke auf Typkorrektheit untersucht. Zur Verbesserung der Übersichtlichkeit schreibt man dabei einen Teilausdruck  $t$  mit Typ  $\tau$  nicht mehr  $t : \tau$ , sondern unterklammert  $t$  und setzt  $\tau$  unter die Klammer,  $\underline{t}_\tau$ .

**Beispiel 4.11 (Typüberprüfung der Formel 3f)**

$$\begin{array}{ccl}
 \mathcal{C}[\text{output } T] \quad \underline{c} & = & \mathcal{T}[T] \quad \lambda n(s, e, a). \quad \underline{c} \quad \langle s, e, a.n \rangle \\
 \underbrace{\text{FORTS.} \longrightarrow \text{FORTS. FORTS.}}_{\text{FORTSETZUNG}} & & \underbrace{\text{TERMF.} \longrightarrow \text{FORTS.}}_{\text{TERMFORT}} \quad \underbrace{\text{FORTS. ZUSTAND}}_{\text{ZUSTAND} + \{\text{Fehler}\}} \\
 & & \underbrace{\text{ZUSTAND}}_{\text{FORTSETZUNG}} \\
 & & \underbrace{\text{TERMFORT}}_{\text{FORTSETZUNG}}
 \end{array}$$

□

Viele Spezifikationsfehler lassen sich auf diese Weise sehr einfach aufdecken. Es gibt auch Algorithmen, mit deren Hilfe sich diese Typüberprüfung automatisch vornehmen lässt (vgl. Damas und Milner [30]).

#### 4.3.4 Modifikation des Ausgabebereiches

Die hier vorgestellte Version der Fortsetzungssemantik der Sprache WHILE behandelt die *Ausgabe* noch nicht zufriedenstellend.

1. Der Ausgabebereich sollte Folgen von Ausgabewerten enthalten, die mit einer Meldung über die Art des Programmstops enden.
2. Der Ausgabebereich sollte keine Zustandskomponente sein, da der Programmrest sowieso keine Zugriffsmöglichkeit auf vorher geschriebene Ausgabewerte hat.
3. Der Ausgabebereich sollte auch unendliche Folgen enthalten, um nichtterminierende Programme, die in einer Endlosschleife Werte ausgeben, vernünftig beschreiben zu können.

Alle drei Korrekturen lassen sich durch folgende Bereichsdefinitionen mit entsprechender Änderung der Semantikfunktionen erreichen (*ANS* steht für „answer“):

$$\begin{aligned} \text{ZUSTAND} &= \text{SPEICHER} \times \text{EINGABE} \\ \text{ANS} &= \{\underline{\text{Fehler}}, \underline{\text{Stop}}\}_{\perp} + (\overline{\text{KON}} \times \text{ANS}) \\ \text{FORTSETZUNG} &= \text{ZUSTAND} \longrightarrow \text{ANS} \end{aligned}$$

(alle anderen Bereiche wie oben).

Jetzt liest sich die semantische Klausel für **output T** wie folgt:

$$\mathcal{C}[\text{output } T] c = T[T] \lambda nz. \langle n, cz \rangle .$$

#### Beispiel 4.12

$$\begin{aligned} &\mathcal{C}[\text{output } 3; \text{output read}] (\lambda z. \underline{\text{Stop}}) \langle S, \langle \rangle \rangle \\ &= \mathcal{C}[\text{output } 3] (\mathcal{C}[\text{output read}] \lambda z. \underline{\text{Stop}}) \langle S, \langle \rangle \rangle \quad \text{nach 3c} \\ &= T[\underline{3}] \lambda nz. \langle n, (\mathcal{C}[\text{output read}] \lambda z. \underline{\text{Stop}}) z \rangle \langle S, \langle \rangle \rangle \quad \text{neue Ausgaberegel} \\ &= \lambda nz. \langle n, (\mathcal{C}[\text{output read}] \lambda z. \underline{\text{Stop}}) z \rangle 3 \langle S, \langle \rangle \rangle \quad \text{nach 1a} \\ &= \langle 3, \mathcal{C}[\text{output read}] (\lambda z. \underline{\text{Stop}}) \langle S, \langle \rangle \rangle \rangle \quad \text{2 } \beta\text{-Reduktionen} \\ &= \langle 3, T[\text{read}] (\lambda nz. \langle n, (\lambda z. \underline{\text{Stop}}) z \rangle \langle S, \langle \rangle \rangle) \rangle \quad \text{neue Ausgaberegel} \\ &= \langle 3, \underline{\text{Fehler}} \rangle \quad \text{nach 1e} \end{aligned}$$

□

#### 4.3.5 Modifikation des Bereiches *SPEICHER*

Der zweite Punkt, der zu Beginn dieses Abschnittes angesprochen wurde, ist die Behandlung gemeinsamer Speicherplätze, englisch *sharing*. Dieses Phänomen gibt es nicht nur in Sprachen mit speziellen Anweisungen der Form

$$x == y ,$$

die bewirken, daß  $x$  und  $y$  auf denselben Speicherplatz verweisen, sondern auch bei allen Prozedur- und Funktionskonzepten, die eine namentliche Parameterübergabe erlauben.

##### Beispiel 4.13 (PASCAL)

```
procedure P(var x : real; var y : real);
begin
  ;
end
...
P(z,z);
...
```

Der Aufruf  $P(z,z)$  bewirkt eine Ausführung der Prozedur  $P$ , bei der die Variablen  $x$ ,  $y$  und  $z$  an denselben Speicherplatz gebunden sind.

□

Zur denotationellen Beschreibung solcher Sachverhalte ist es zweckmäßig, die Zuordnungsfunktion zur Bestimmung des Wertes einer Variablen in zwei Abbildungen aufzuspalten. Dann ist es möglich, mit der ersten die Adresse eines Speicherplatzes zu ermitteln und mit der zweiten den Inhalt dieses Speicherplatzes. Der Bereich

$$SPEICHER = ID \longrightarrow (Z \perp + \{frei\} \perp)$$

wird durch die beiden Bereiche

$$\begin{aligned} ENV &= ID \longrightarrow (LOC + \{\underline{ungebunden}\} \perp) \text{ und} \\ STORE &= LOC \longrightarrow (Z \perp + \{frei\} \perp) . \end{aligned}$$

ersetzt. Im allgemeinen Fall definiert man die neuen Semantikfunktionen so, daß

$$SPEICHER I = STORE(ENV I)$$

für alle  $I \in ID$  gilt.

$ENV$  ist der semantische Bereich der *Umgebungen* (*environments*),  $STORE$  der Bereich der *adressierbaren Speicher* und  $LOC$  der Bereich der *Adressen der Speicherplätze* (*locations*). Nun ist leicht einzusehen, wie zwei Variablen  $x$  und  $y$  einen Speicherplatz teilen können: Man erzeugt eine neue Umgebung  $\rho$ , so daß  $\rho x = \rho y$

gilt. Die Ausführung einer Wertzuweisung ändert nur den Speicher (STORE), nicht aber die Umgebung (ENV). Angenommen  $\rho x = l \in LOC$ , und der aktuelle Speicher sei  $s \in STORE$ , dann überführt die Ausführung der Anweisung  $x := 7$  diesen Speicher in  $s[7/l]$ . Definiert man nun  $\llbracket I \rrbracket \rho s = s(\rho I)$ , so gilt:

$$\llbracket x \rrbracket \rho s = s(\rho x) = sl = 7$$

und auch

$$\llbracket y \rrbracket \rho s = s(\rho y) = sl = 7 .$$

Dies ist eine vereinfachte Form der direkten Semantik, die weder die Fälle  $\rho I = \underline{\text{ungebunden}}$  oder  $s(\rho I) = \underline{\text{frei}}$  berücksichtigt noch eine entsprechende Fortsetzung verwendet. Eine ausführliche Fassung wird im Zusammenhang mit der Sprache *PASCAL<sub>0</sub>* in Abschnitt 4.4 behandelt.

I.a. nennt man eine denotationelle Semantikspezifikation, die sowohl mit Fortsetzungen arbeitet als auch das Umgebungs-/Speicherkonzept verwendet, *Standardsemantik*.

#### 4.3.6 Standardwertebereiche

Für eine Sprache, die im Rahmen der Standardsemantik beschrieben werden soll, lassen sich mindestens drei *Standardwertebereiche* angeben:

1. Die *ausdrückbaren Werte EV (expressible values)* - dies sind diejenigen Werte, die durch eine Berechnung von Termen gewonnen werden können. Also

$$EV = \overline{KON} + \dots .$$

2. Die *bezeichnenbaren Werte DV (denotable values)* - dies sind diejenigen Werte, die in der Umgebung an eine Variable (einen Bezeichner) gebunden werden können. Also

$$DV = LOC + \dots .$$

3. Die *speicherbaren Werte SV (storable values)* - dies sind diejenigen Werte, die im Speicher abgelegt werden können. Also

$$SV = \overline{KON} + \dots .$$

Die Angabe der Standardwertebereiche einer Programmiersprache legt bereits einige ihrer Eigenschaften fest. So bedeutet etwa

$$EV = \mathbb{Z}_\perp + \overline{BOOL} ,$$

dass alle Terme über ganzen Zahlen und booleschen Werten interpretiert werden, und

$$DV = LOC + \overline{KON} ,$$

daß in der Umgebung sowohl Adressen von Speicherplätzen als auch konstante Werte an die Bezeichner gebunden werden können, und schließlich bedeutet

$$SV = \overline{KON} + LOC + PROC ,$$

daß im Speicher sowohl Konstanten als auch Adressen als auch Prozeduren abgelegt werden können.

In der Sprache PASCAL z.B. sind die Konstanten bezeichbar und die Adressen der Speicherplätze speicherbar. Beides gilt nicht für die Sprache ALGOL 60.

#### 4.3.7 Deklarationen

Wie bereits oben erläutert, ändert die Ausführung einer Anweisung den aktuellen Speicher, also das entsprechende Objekt aus *STORE*. Man braucht aber auch eine Möglichkeit, die aktuelle Umgebung zu modifizieren. Dazu dienen die *Deklarationen*, die zu Beginn eines Blockes oder einer Prozedur auftreten.

##### Beispiel 4.14

```

begin
    integer x; boolean y;
    :
    x := 7;
    y := false ;
    :
end

```

*Bei Eintritt in diesen Block werden den Variablen x und y in der aktuellen Umgebung neue Speicherplätze zugeordnet, die während des gesamten Blockes konstant bleiben. Alle Anweisungen innerhalb des Blockes verändern nur die Inhalte der Speicherplätze.*

□

Wie man bereits an obigem Beispiel erkennen kann, ist ein Mechanismus erforderlich, der zu einem gegebenen Speicher die Adresse eines neuen Speicherplatzes auswählt, falls noch ein freier Speicherplatz vorhanden ist und der ansonsten eine Fehlermeldung produziert. Meist wird bei einer denotationellen Semantikspezifikation angenommen, daß eine Funktion

$$\underline{new} : STORE \longrightarrow (LOC + \{\underline{\text{Fehler}}\})$$

mit folgenden Eigenschaften für alle  $s \in STORE$  existiert:

1. Wenn  $\underline{isLoc}(\underline{new} s) = \text{wahr}$  gilt, dann ist  $s(\underline{new} s) = \underline{\text{frei}}$ .

2. Wenn *new*  $s = \text{Fehler}$  gilt, dann gilt  $sl \neq \text{frei}$  für alle  $l \in LOC$ .

Oft wird die Funktion *new* nicht näher spezifiziert. I.a. ist es jedoch nicht ausreichend, eine Funktion nur durch ihre Eigenschaften zu charakterisieren. Daher soll hier eine explizite Definition unter der Annahme, daß  $LOC = \{l_1, l_2, \dots, l_n\}$  ist, angegeben werden:

$$\begin{aligned} \underline{\text{new}} &= \lambda s. \quad sl_1 = \underline{\text{frei}} \rightarrow l_1, \\ &\quad sl_2 = \underline{\text{frei}} \rightarrow l_2, \\ &\quad \vdots \\ &\quad sl_n = \underline{\text{frei}} \rightarrow l_n, \\ &\quad \underline{\text{Fehler}} \end{aligned}$$

Nun lassen sich auf einer informellen Ebene bereits verschiedene Deklarationstypen betrachten:

**const**  $I = T$

bewirkt eine Änderung der Umgebung derart, daß dem Bezeichner  $I$  direkt der Wert von  $T$  zugeordnet wird, d.h. eine solche Deklaration verlangt  $DV = \dots + \overline{KON} + \dots$ .

**var**  $I = T$

erfordert die Adresse  $l$  eines neuen Speicherplatzes, belegt diesen mit dem Wert von  $T$  und bindet schließlich in der aktuellen Umgebung die Variable  $I$  an die Adresse  $l$ .

**procedure**  $P(I); C$

bindet in der aktuellen Umgebung die Variable  $P$  an eine Prozedur mit einem Parameter  $I$  und Prozedurrumpf  $C$ .

**function**  $F(I); T$

bindet in der aktuellen Umgebung die Variable  $F$  an eine Funktionsprozedur mit einem Parameter  $I$  und Funktionswert, bestimmt durch  $T$ .

$D_1; D_2$

bewirkt eine Umgebungsänderung gemäß der Deklaration  $D_1$  gefolgt von der Umgebungsänderung gemäß der Deklaration  $D_2$ .

### 4.3.8 Standardfortsetzungsfunktionen

Ähnlich wie die Standardwertebereiche lassen sich auch die *Standardfortsetzungsfunktionen* beschreiben. Dabei unterscheidet man drei Typen:

1. die *Anweisungsfortsetzungen CC* (*command continuations*),
2. die *Ausdrucksfortsetzungen EC* (*expression continuations*) und
3. die *Deklarationsfortsetzungen DC* (*declaration continuations*).

Diese Fortsetzungsfunktionen sind von folgenden Typen:

$$\begin{aligned} CC &= STORE \rightarrow ANS, \\ EC &= EV \rightarrow CC \text{ und} \\ DC &= ENV \rightarrow CC. \end{aligned}$$

Eine Standardsemantikfunktion definiert folgende Semantikfunktionen:

$$\begin{aligned} C &: COM \rightarrow ENV \rightarrow CC \rightarrow STORE \rightarrow ANS, \\ E &: EXP \rightarrow ENV \rightarrow EC \rightarrow STORE \rightarrow ANS \text{ und} \\ D &: DEC \rightarrow ENV \rightarrow DC \rightarrow STORE \rightarrow ANS, \end{aligned}$$

wobei hier alle Ausdrücke syntaktisch zu einer Menge  $EXP$  zusammengefaßt sind, d.h.

$$EXP = TERM + BT + \dots$$

Nun ist es die Aufgabe der Semantikfunktion  $E$ , die Typen der Ausdrücke zu überprüfen.

Wenn keine irregulären Kontrollverläufe auftreten, lassen sich die Semantikfunktionen wie folgt definieren:

$C[C]_{\rho s} = cs'$ , wobei  $s'$  derjenige Speicher ist, der durch Ausführung von  $C$  in der Umgebung  $\rho$  aus dem Speicher  $s$  entsteht.

$E[E]_{\rho ks} = kes'$ , wobei  $e$  der Wert von  $E$  in der Umgebung  $\rho$  und bzgl. dem Speicher  $s$  ist und  $s'$  der Speicher, der durch Berechnung von  $e$  aus  $s$  entsteht.

$D[D]_{\rho us} = u\rho's'$ , wobei  $\rho'$  die durch  $D$  definierte Umgebung ist und  $s'$  derjenige Speicher, der durch die Deklaration  $D$  aus  $s$  entsteht.

Jede Semantikfunktion übergibt also im Normalfall nach Anwendung auf eine syntaktische Klausel ihre Ergebnisse an ihr drittes Argument, welches gerade die entsprechende Fortsetzung ist.

#### 4.3.9 Fortsetzungstransformationen

Gewisse Hilfsfunktionen werden im Rahmen der Standardsemantik gerne als *Fortsetzungstransformationen* definiert, um sie bei der Definition der Semantikfunktionen elegant verwenden zu können. Als Beispiel dafür seien hier die Hilfsfunktionen, die im Umgang mit dem neuen Speicherkonzept nötig sind, vorgestellt. Es handelt sich um

- eine Funktion cont, die den Inhalt eines Speicherplatzes ermittelt,
- eine Funktion update, die einen Speicherplatz neu belegt,

- eine Funktion *ref*, die den Wert eines Ausdruckes in einem neuen Speicherplatz ablegt,
- eine Funktion *deref*, die einen Wert dereferenziert, d.h. angewendet auf eine Konstante, wird diese übergeben, aber angewendet auf eine Adresse, wird der Inhalt des entsprechenden Speicherplatzes übergeben, und schließlich
- eine Funktion *D?* zu jedem Bereich *D*, die ein Argument nur dann weiterreicht, wenn es aus dem Bereich *D* ist, und andernfalls eine Fehlermeldung erzeugt.

Formal sind diese Funktionen wie folgt definiert:

- *cont* : *EC* → *EC*  
 $\underline{\text{cont}}\ k = \lambda e. \underline{\text{isLoc}} e \rightarrow (se - \underline{\text{frei}} \rightarrow \underline{\text{Fehler}}, k(se)s), \underline{\text{Fehler}}$
- *update* : *LOC* → *CC* → *EC*  
 $\underline{\text{update}}\ lc = \lambda s. \underline{\text{issV}} e \rightarrow cs[e/l], \underline{\text{Fehler}}$
- *ref* : *EC* → *EC*  
 $\underline{\text{ref}}\ k = \lambda e. \underline{\text{new}}\ s = \underline{\text{Fehler}} \rightarrow \underline{\text{Fehler}}, \underline{\text{update}}(\underline{\text{new}}\ s)(k(\underline{\text{new}}\ s))es$
- *deref* : *EC* → *EC*  
 $\underline{\text{deref}}\ k = \lambda e. \underline{\text{isLoc}} e \rightarrow \underline{\text{cont}}\ kes, kes$
- *D?* : *EC* → *EC*  
 $\underline{\text{D?}}\ k = \lambda e. \underline{\text{isD}} e \rightarrow kes, \underline{\text{Fehler}}$

Mit der speziellen Fehlerfortsetzung

- *err* : *CC*  
 $\underline{\text{err}} = \lambda s. \underline{\text{Fehler}}$

lässt sich die Funktion *D?* noch vereinfachen zu:

- $\underline{\text{D?}}\ k = \lambda e. \underline{\text{isD}} e \rightarrow ke, \underline{\text{err}}$

#### 4.3.10 Verallgemeinerung der Wertzuweisung

Bisher wurde nur eine einfache Form der Wertzuweisung behandelt, nämlich

$$I := E .$$

In einigen Programmiersprachen sind jedoch auch auf der linken Seite der Zuweisung allgemeinere Ausdrücke erlaubt.

**Beispiel 4.15**

$$\begin{aligned} x &:= \text{if } a > b \text{ then } a \text{ else } b; \\ (\text{if } a > b \text{ then } x \text{ else } y) &:= c \end{aligned}$$

□

Man erkennt an obigem Beispiel, daß auf beiden Seiten der Wertzuweisung Ausdrücke von gleicher syntaktischer Struktur auftreten, so daß die entsprechende Syntaxregel

$$E_1 := E_2$$

lauten könnte. Semantisch müssen jedoch Ausdrücke auf der linken und rechten Seite einer Zuweisung unterschiedlich behandelt werden. Nennen wir den Wert, den die Berechnung des *linken* Ausdruckes liefert, L-Wert und denjenigen des *rechten* Ausdruckes R-Wert, dann soll natürlich der L-Wert eine Adresse sein und der R-Wert ein speicherbarer Wert. Dieser unterschiedlichen Semantik gleichartiger Ausdrücke kann man nur durch zwei verschiedene Semantikfunktionen Rechnung tragen. Anstelle von  $\mathcal{E}$  verwendet man die Funktion  $\mathcal{R}$  bzw.  $\mathcal{L}$  aus dem Bereich  $EXP \rightarrow ENV \rightarrow EC \rightarrow CC$ .

$$\begin{aligned} \mathcal{R}[E]\rho k &= \mathcal{E}[E]\rho(\underline{\text{deref}}\ k) \\ \mathcal{L}[E]\rho k &= \mathcal{E}[E]\rho(\underline{\text{LOC?}}\ k) \end{aligned}$$

Mit diesen beiden Funktionen kann nun die Semantikfunktion für die verallgemeinerte Wertzuweisung recht einfach definiert werden:

$$\mathcal{C}[E_1 := E_2]\rho c = \mathcal{L}[E_1]\rho \lambda i. \mathcal{R}[E_2]\rho; \underline{\text{update}}\ l; c.$$

Eine Beispielrechnung dazu findet man im Anschluß an die Definition der Semantik von *PASCAL*<sub>0</sub> in Abschnitt 4.4.

#### 4.3.11 Standardsemantik von Prozeduren und Funktionen

Das letzte Thema, das im Rahmen der Entwicklung der Standardsemantik angesprochen werden soll, ist die semantische Beschreibung eines einfachen Prozedur- und Funktionsprozedurkonzeptes. Es sollen geeignete semantische Bereiche *PROC* und *FUN* konstruiert werden, die die Semantik einfacher Prozeduren bzw. Funktionen, wie sie im Zusammenhang mit Deklarationen in Unterabschnitt 4.3.7 vorgestellt wurden, enthalten können.

Syntaktisch lauten die Prozedur- und Funktionsprozedurdeklarationen

**procedure  $P(I); C$  bzw. function  $F(I); E$ .**

Die Einschränkung auf genau einen Parameter sei hier gemacht, um die Bereichsdefinitionen zunächst möglichst einfach zu halten. Die Behandlung verallgemeinerter

Prozedurkonzepte findet man in Abschnitt 4.5.

Die Wirkung von Aufrufen der Form  $P(E_1)$  bzw.  $F(E_1)$  läßt sich wie folgt beschreiben: Zunächst soll der Ausdruck  $E_1$  bei vorliegendem Speicher  $s$  ausgewertet werden. Das Ergebnis davon ist ein ausdrückbarer Wert  $e$  und ein veränderter Speicher  $s'$ . Sodann kann bei Speicher  $s'$  die Anweisung  $C$  ausgeführt bzw. der Ausdruck  $E$  berechnet werden, wobei während dieses Vorganges die Bindung von  $I$  an den Wert  $e$  gültig ist. Die Antwort auf die Frage, ob an die Variable  $I$  der R- oder L-Wert von  $E_1$  gebunden wird, hängt von dem vereinbarten Parameterübergabemodus ab. Hier soll zunächst die einfache Semantikfunktion  $\mathcal{E}$  zur Berechnung der aktuellen Parameter verwendet werden, d.h.  $e$  ist der Wert von  $E_1$  bzgl. der Bindungen der auftretenden Variablen in der aktuellen Umgebung. Weitere Übergabemechanismen werden ebenfalls in Abschnitt 4.5 untersucht.

Der Bereich  $PROC$  soll nun so konstruiert werden, daß seine Objekte die Semantik von Prozeduren geeignet repräsentieren. Ein Element aus dem Bereich  $PROC$  soll bei gegebener Fortsetzung (Semantik des Programmrestes hinter einem Aufruf) zu einem ausdrückbaren Wert (dem aktuellen Parameter) und einem vorliegenden Speicher eine neue Ausgabe bestimmen, d.h.

$$PROC = CC \longrightarrow EV \longrightarrow STORE \longrightarrow ANS$$

oder abgekürzt

$$PROC = CC \longrightarrow EC .$$

Nun lautet die Semantik der Prozedurdeklaration:

$$\begin{aligned} \mathcal{D}[\text{procedure } P(I); C]_{\rho u} &= u[p/P], \text{ wobei} \\ p &= \lambda ce. \mathcal{C}[C]_{\rho[e/I]c} . \end{aligned}$$

Die Semantik eines Prozedurauftrufes kann man jetzt wie folgt formalisieren:

$$\mathcal{C}[P(E)]_{\rho cs} = \mathcal{E}[E]_{\rho} \lambda es. \rho P ces .$$

Um eine Fehlermeldung zu erzeugen für den Fall, daß die Variable  $P$  in der Umgebung  $\rho$  nicht an einen Prozedurwert aus  $PROC$  gebunden ist, kann die Semantik eines Aufrufes noch verbessert werden zu:

$$\mathcal{C}[P(E)]_{\rho c} = \mathcal{E}[P]_{\rho}; PROC? \lambda p. \mathcal{E}[E]_{\rho}; pc .$$

Diese Definition läßt sich lesen als: Bestimme den Wert von  $P$  in der Umgebung  $\rho$ . Wenn dies ein Prozedurwert  $p$  aus dem Bereich  $PROC$  ist, reiche ihn weiter und berechne die Semantik von  $E$  in der Fortsetzung  $pc$ .

Diese Konstruktion läßt sich analog für Funktionsprozeduren durchführen:

$$FUN = EC \longrightarrow EC .$$

$$\begin{aligned} \mathcal{D}[\text{function } F(I); E]_{\rho u} &= u[f/F], \text{ wobei} \\ f &= \lambda ke. \mathcal{E}[E]_{\rho[e/I]k} . \end{aligned}$$

Die Semantik eines Funktionsaufrufes lautet entsprechend:

$$\mathcal{E}[\![F(E)]\!]_{\rho k} = \mathcal{E}[\![F]\!]_{\rho}; \text{FUN? } \lambda f. \mathcal{E}[\![E]\!]_{\rho}; fk.$$

Dies ist eine sehr einfache Form von Funktionsprozeduren. Normalerweise besteht eine solche aus einer Anweisung  $C$ , gefolgt von einem Ausdruck  $E$ , dessen Wert nach Ausführung von  $C$  den Wert eines Aufrufes liefert. Eine semantische Behandlung dieser Verallgemeinerung findet sich in Abschnitt 4.5.

Mit diesen Definitionen endet hier die Beschreibung der Methode der Standardsemantik. Die denotationelle Semantik einer beliebigen, sequentiellen Programmiersprache kann nun vollständig und einheitlich im Rahmen der Standardsemantik beschrieben werden. Damit hat man ein geeignetes Werkzeug zur Definition und zum Vergleich von Programmiersprachen entwickelt.

## 4.4 Die Standardsemantik der Sprache $PASCAL_0$

In diesem Abschnitt sollen alle bisher entwickelten Techniken anhand einer umfangreicheren Sprache,  $PASCAL_0$ , angewendet werden. Diese Sprache wird im Rahmen der Standardsemantik vollständig spezifiziert.  $PASCAL_0$  ist im wesentlichen eine eingeschränkte Version der von Wirth entwickelten Sprache  $PASCAL$  (siehe Jensen und Wirth [55]).

Es wird zunächst eine kompakte Sprachdefinition vorgestellt. Anschließend werden einige Erläuterungen gegeben und eine Beispielberechnung durchgeführt. Die Sprache  $PASCAL_0$  wird für alle späteren Betrachtungen stets denjenigen Rahmen darstellen, in den alle weiteren Konzepte eingebettet werden.

### 4.4.1 Syntax von $PASCAL_0$

#### Syntaktische Bereiche mit dazugehörigen Metavariablen

##### Primitive Bereiche

$KON$  Bereich aller Konstanten  $K$

$ID$  Bereich aller Bezeichner  $I, P, F$

$OP$  Bereich aller 2-stelligen Operationen  $O$

##### Zusammengesetzte Bereiche

$EXP$  Bereich aller Ausdrücke  $E$

$COM$  Bereich aller Anweisungen  $C$

$DEC$  Bereich aller Deklarationen  $D$

$PROG$  Bereich aller Programme  $P$

Syntaktische Klauseln

$E ::= K \mid I \mid E_1 \circ E_2 \mid \text{read } | E_1(E_2) \mid \text{if } E \text{ then } E_1 \text{ else } E_2$   
 $C ::= \text{skip } \mid E_1 := E_2 \mid C_1 \cdot C_2 \mid \text{if } E \text{ then } C_1 \text{ else } C_2 \mid \text{while } E \text{ do } C \mid$   
 $\quad \text{write } E \mid E_1(E_2) \mid \text{begin } D; C \text{ end}$   
 $D ::= \text{const } I = E \mid \text{var } I = E \mid \text{procedure } P(I); C \mid \text{function } F(I); E \mid$   
 $\quad D_1; D_2$   
 $P ::= C$

**4.4.2 Semantik von PASCAL<sub>0</sub>**Semantische Bereiche mit dazugehörigen Metavariablen

$LOC$	Bereich der Speicherplatzadressen	$l$
$BV$	$= \mathbb{Z}_\perp + \overline{BOOL} + \dots$	Basiswerte $e$
$DV$	$= LOC + RV + PROC + FUN$	bezeichbare Werte $d$
$SV$	$= FILE + RV$	speicherbare Werte $v$
$EV$	$= DV$	ausdrückbare Werte $e$
$RV$	$= BV$	R-Werte $e$
$FILE$	$= RV^*$	Dateien (Files) $f$
$ENV$	$= ID \longrightarrow (DV + \{\underline{\text{ungebunden}}\}_\perp)$	Umgebungen $\rho$
$STORE$	$= LOC \longrightarrow (SV + \{\underline{\text{frei}}\}_\perp)$	Speicher $s$
$CC$	$= STORE \longrightarrow ANS$	Anweisungsfortsetzungen $c$
$EC$	$= EV \longrightarrow CC$	Ausdrucksfortsetzungen $k$
$DC$	$= ENV \longrightarrow CC$	Deklarationsfortsetzungen $u$
$PROC$	$= CC \longrightarrow EC$	Prozedurwerte $p$
$FUN$	$= EC \longrightarrow EC$	Funktionswerte $f$
$ANS$	$= \{\underline{\text{Fehler}}, \underline{\text{Stop}}\}_\perp + (RV \times ANS)$	Ausgaben $a$

Annahme: Der Bereich der Adressen  $LOC$  enthält einen Speicherplatz  $\underline{\text{input}}$ , dessen Inhalt die Eingabedatei ist.

### Semantikfunktionen

Gegeben seien die Funktionen

$$\begin{aligned}\mathcal{B} &: KON \longrightarrow BV \text{ und} \\ \mathcal{O} &: OP \longrightarrow (RV \times RV) \longrightarrow EC \longrightarrow CC.\end{aligned}$$

Nichtelementare Funktionen sind:

$$\begin{aligned}\mathcal{E} &: EXP \longrightarrow ENV \longrightarrow EC \longrightarrow CC \\ \mathcal{R} &: EXP \longrightarrow ENV \longrightarrow EC \longrightarrow CC \\ \mathcal{C} &: COM \longrightarrow ENV \longrightarrow CC \longrightarrow CC \\ \mathcal{D} &: DEC \longrightarrow ENV \longrightarrow DC \longrightarrow CC \\ \mathcal{P} &: PROG \longrightarrow FILE \longrightarrow ANS\end{aligned}$$

### Semantische Klauseln

#### 1. Semantik der Ausdrücke

- (a)  $\mathcal{E}[K]\rho k = k(\mathcal{B}[K])$
- (b)  $\mathcal{E}[I]\rho k = (\rho I = \underline{\text{ungebunden}}) \longrightarrow \text{err}, k(\rho I)$
- (c)  $\mathcal{E}[E_1 \ O \ E_2]\rho k = \mathcal{R}[E_1]\rho \lambda e_1. \mathcal{R}[E_2]\rho \lambda e_2. \mathcal{O}[O]\langle e_1, e_2 \rangle k$
- (d)  $\mathcal{E}[\text{read } s]\rho k = \underline{\text{null}}(s \ \underline{\text{input}}) \longrightarrow \underline{\text{Fehler}},$   
 $k(hd(s \ \underline{\text{input}})) \ s[tl(s \ \underline{\text{input}})/\underline{\text{input}}]$
- (e)  $\mathcal{E}[E_1(E_2)]\rho k = \mathcal{E}[E_1]\rho; \text{FUN? } \lambda f. \mathcal{E}[E_2]\rho; \ f k$
- (f)  $\mathcal{E}[\text{if } E \text{ then } E_1 \text{ else } E_2]\rho k = \mathcal{R}[E]\rho; \ \overline{\text{BOOL? cond}}(\mathcal{E}[E_1]\rho k, \mathcal{E}[E_2]\rho k)$

#### 2. R-Semantik der Ausdrücke

- (a)  $\mathcal{R}[E]\rho k = \mathcal{E}[E]\rho; \ \underline{\text{deref}}; \ RV? k$

#### 3. Semantik der Anweisungen

- (a)  $\mathcal{C}[\text{skip}]\rho c = c$
- (b)  $\mathcal{C}[E_1 := E_2]\rho c = \mathcal{E}[E_1]\rho; \ LOC? \lambda l. \mathcal{R}[E_2]\rho; \ \underline{\text{update}} \ l \ c$

- (c)  $\mathcal{C}[[C_1; C_2]]_{\rho c} = \mathcal{C}[[C_1]]_{\rho}; \mathcal{C}[[C_2]]_{\rho c}$
- (d)  $\mathcal{C}[[\text{if } E \text{ then } C_1 \text{ else } C_2]]_{\rho c} =$   
 $\mathcal{R}[[E]]_{\rho}; \overline{\text{BOOL}}?; \underline{\text{cond}}(\mathcal{C}[[C_1]]_{\rho c}, \mathcal{C}[[C_2]]_{\rho c})$
- (e)  $\mathcal{C}[[\text{while } E \text{ do } C]]_{\rho c} =$   
 $\mathcal{R}[[E]]_{\rho}; \overline{\text{BOOL}}?; \underline{\text{cond}}(\mathcal{C}[[C; \text{while } E \text{ do } C]]_{\rho c}, c)$
- (f)  $\mathcal{C}[[\text{write } E]]_{\rho c} = \mathcal{R}[[E]]_{\rho} \lambda e.s.(e, cs)$
- (g)  $\mathcal{C}[[E_1(E_2)]]_{\rho c} = \mathcal{E}[[E_1]]_{\rho}; \text{PROC? } \lambda p.\mathcal{E}[[E_2]]_{\rho}; pc$
- (h)  $\mathcal{C}[[\text{begin } D; C \text{ end }]]_{\rho c} = \mathcal{D}[[D]]_{\rho} \lambda \rho'. \mathcal{C}[[C]]_{\rho[\rho']} c$

#### 4. Semantik der Deklarationen

- (a)  $\mathcal{D}[[\text{const } I = E]]_{\rho u} = \mathcal{R}[[E]]_{\rho} \lambda e.u(e/I)$
- (b)  $\mathcal{D}[[\text{var } I = E]]_{\rho u} = \mathcal{R}[[E]]_{\rho}; \underline{\text{ref}} \lambda l.u(l/I)$
- (c)  $\mathcal{D}[[\text{procedure } P(I); C]]_{\rho u} = u(\lambda ce.\mathcal{C}[[C]]_{\rho[e/I] c/P})$
- (d)  $\mathcal{D}[[\text{function } F(I); E]]_{\rho u} = u(\lambda ke.\mathcal{E}[[E]]_{\rho[e/I] k/F})$
- (e)  $\mathcal{D}[[D_1; D_2]]_{\rho u} = \mathcal{D}[[D_1]]_{\rho} \lambda \rho_1. \mathcal{D}[[D_2]]_{\rho[\rho_1]} \lambda \rho_2. u(\rho_1[\rho_2])$

#### 5. Semantik der Programme

$$\mathcal{P}[[C]] i = \mathcal{C}[[C]]() (\lambda s. \underline{\text{Stop}})(i/\underline{\text{input}})$$

#### 4.4.3 Bemerkungen zur Definition von PASCAL<sub>0</sub>

##### Primitive und zusammengesetzte syntaktische Bereiche

Ein syntaktischer Bereich mit beliebig mächtiger und strukturierter Grundmenge ist aus der Sicht der Standardsemantik genau dann primitiv, wenn die dazugehörige Semantikfunktion in direkter Weise für jedes seiner Objekte definiert ist.

**Beispiel 4.16**

- Der Bereich  $NUM$ , der aus dem Nichtterminal  $N$  mit der BNF-Definition

$$N = 0 \mid 1 \mid 2 \mid \dots$$

erzeugt wird, ist primitiv, wenn die entsprechende Semantikfunktion durch eine Gleichung definiert ist, z.B.

$$\mathcal{E}[n]\rho = n \text{ für alle } n \in IN .$$

Bei einer solchen Definition geht man davon aus, daß die Zuordnung von Zahldarstellungen zu ihren semantischen Werten elementar ist, und unterscheidet häufig syntaktische Objekte in der Schreibweise nicht mehr von den entsprechenden semantischen Werten.

- Der Bereich  $NUM$  mit einer BNF-Definition der Form

$$N = 0 \mid 1 \mid N0 \mid N1$$

ist ein zusammengesetzter Bereich, wenn die Semantikfunktion induktiv über den Aufbau einer Zahl definiert ist, z.B.

$$\begin{aligned}\mathcal{E}[0]\rho &= 0 \\ \mathcal{E}[1]\rho &= 1 \\ \mathcal{E}[N0]\rho &= 2 \cdot \mathcal{E}[N]\rho \\ \mathcal{E}[N1]\rho &= 2 \cdot \mathcal{E}[N]\rho + 1\end{aligned}$$

Bei dieser Definition wird also nur die Abbildung von 0 und 1 auf 0 bzw. 1 als elementar vorausgesetzt.

□

**Syntaktische Klauseln**

Die meisten Syntaxformen sind bereits von der Sprache WHILE her bekannt. Anstatt wie bisher zwischen arithmetischen Ausdrücken, booleschen Termen u.a. zu unterscheiden, wird in  $PASCAL_0$  nur eine Menge von Ausdrücken betrachtet, so daß die Typinformation syntaktisch verlorengeht, d.h. beliebige Konstruktionen wie

true + 4 oder if 3 - 7 then  $C_1$  else  $C_2$

sind zwar syntaktisch erlaubt, werden aber von der Semantikfunktion  $\mathcal{O}$  als fehlerhaft erkannt.

Neu ist auch die allgemeine Form der Prozedur- und Funktionsaufrufe. Anstelle von  $P(E)$  und  $F(E)$  werden beliebige Ausdrücke der Form  $E_1(E_2)$  zugelassen. Dies kann motiviert werden durch die Überlegung, daß es manchmal sinnvoll ist, einen Ausdruck zu verwenden, der sich zu einem Prozedur- oder Funktionsnamen

berechnet, z.B. wenn der aktuelle Parameter für einen Aufruf bestimmt ist, die anzuwendende Prozedur jedoch vom aktuellen Speicher abhängt, etwa:

$$(\text{if } I = 1 \text{ then } P \text{ else } Q)(7).$$

### Semantische Bereiche

Der Grund, warum die Bereiche  $EV$  und  $RV$  explizit definiert sind, anstatt überall dort, wo ihre Namen vorkommen, entsprechend  $DV$  bzw.  $BV$  einzusetzen, liegt darin, daß man zum Ausdruck bringen möchte, daß in dieser Sprache alle bezeichnaren Werte auch ausdrückbar sind und alle Basiswerte auch Rechtswerte sind. Man möchte dennoch diese Bereiche konzeptuell unterscheiden, d.h. wenn ein Bezeichner  $x$  in der Umgebung  $\rho$  an die Adresse  $l$  gebunden ist, so ist  $x$  in diesem Fall ein bezeichnbarer Wert, und wenn  $x$  als Ausdruck auf der rechten Seite einer Anweisung der Form  $I := x$  vorkommt und sich die Semantik  $\mathcal{E}[x]_\rho$  zu  $l$  berechnet, dann ist  $l$  in diesem Fall ein ausdrückbarer Wert.

### Semantikfunktionen

Die Semantikfunktion  $\mathcal{O}$  für die zweistelligen Operationen würde man zunächst im Bereich

$$OP \longrightarrow (RV \times RV) \longrightarrow RV$$

spezifizieren. Mit dem hier angegebenen Typ

$$OP \longrightarrow (RV \times RV) \longrightarrow EC \longrightarrow CC$$

wird jedoch gleich die Semantik der Operationssymbole als fortsetzungstransformierende Funktionen einheitlich dargestellt.

### Semantische Klauseln

Für Umgebungen und Speicher wird eine Standardabkürzung verwendet, die einer Initialisierung aller nicht spezifizierten Variablen mit ungebunden bzw. frei entspricht.

Insbesondere definiert man:

$$\begin{aligned} d_1, \dots, d_n / I_1, \dots, I_n &= \lambda I. \quad I = I_1 \longrightarrow d_1, \\ &\quad I = I_2 \longrightarrow d_2, \\ &\quad \vdots \\ &\quad I = I_n \longrightarrow d_n, \\ &\quad \underline{\text{ungebunden}} \end{aligned}$$

und

$$\begin{aligned}
 v_1, \dots, v_n / l_1, \dots, l_n &= \lambda l. \quad l = l_1 \longrightarrow v_1, \\
 &\quad l = l_2 \longrightarrow v_2, \\
 &\quad \vdots \\
 &\quad l = l_n \longrightarrow v_n, \\
 &\quad \underline{\text{frei}}
 \end{aligned}$$

für alle  $I_i \in ID$ ,  $d_i \in DV$  und  $v_i \in SV$  mit  $1 \leq i \leq n$ ,  $n \in IN$ . Insbesondere gilt:

$$() \in ENV \text{ mit } () = \lambda I. \underline{\text{ungebunden}}$$

und

$$() \in STORE \text{ mit } () = \lambda l. \underline{\text{frei}} .$$

Diese Schreibweise lässt sich mit derjenigen zur Modifikation von Funktionen (vgl. Abschnitt 4.1.5) so kombinieren, daß aus Umgebungen  $\rho$  und  $\rho'$  die *erweiterte Umgebung*  $\rho[\rho']$  entsteht, die durch die Gleichung

$$\rho[\rho'] = \lambda I. (\rho' I = \underline{\text{ungebunden}}) \longrightarrow \rho I, \rho' I$$

definiert ist.

#### 4.4.4 Berechnung der denotationellen Semantik eines Beispielprogramms

Die semantischen Klauseln der Sprache  $PASCAL_0$  sollen nun anhand eines kleinen Beispiels angewendet werden.

Gegeben sei folgendes  $PASCAL_0$ -Programm:

$$P \left\{ \begin{array}{l} \mathbf{begin} \\ D \{ \mathbf{var} \ x = 1; \\ C \{ C_1 \{ x := x + 2; \\ \quad C_2 \{ \mathbf{write} \ x \\ \mathbf{end} \end{array} \right.$$

Die Semantik von  $P$ , angewendet auf die Eingabedatei  $\langle \rangle$ , lässt sich wie folgt berechnen:

$$\begin{aligned}
& \mathcal{P}[P] \langle \rangle \\
= & \mathcal{C}[P](\lambda s. \underline{Stop})(\langle \rangle / \underline{input}) & 5 \\
= & \mathcal{D}[D](\lambda \rho. \mathcal{C}[C] \rho \lambda s. \underline{Stop})(\langle \rangle / \underline{input}) \\
= & \mathcal{R}[1](\underline{ref} \lambda l. u(l/x)) (\langle \rangle / \underline{input}) & 4b \\
= & \mathcal{E}[1](\underline{deref}; \underline{RV?}; \underline{ref} \lambda l. u(l/x)) (\langle \rangle / \underline{input}) & 2a \\
= & (\underline{deref}; \underline{RV?}; \underline{ref} \lambda l. u(l/x)) 1 (\langle \rangle / \underline{input}) & 1a \\
= & (\underline{ref} \lambda l. u(l/x)) 1 (\langle \rangle / \underline{input}) & \text{Def. von } \underline{deref} \text{ und } \underline{RV?} \\
= & \underline{update} l_1 u(l_1/x) 1 (\langle \rangle / \underline{input}) & \text{Def. von } \underline{ref} \text{ und } \beta\text{-Red.} \\
= & \underline{is}_{SV} 1 \xrightarrow{*} u(l_1/x) (\langle \rangle, 1 / \underline{input}, l_1), \underline{Fehler} & \text{Def. von } \underline{update} \\
= & u(l_1/x) s & \text{Def. von } SV \\
= & (\lambda \rho. \mathcal{C}[C] \rho \lambda s. \underline{Stop})(l_1/x) s \\
= & \mathcal{C}[C](l_1/x) (\lambda s. \underline{Stop}) s & \beta\text{-Reduktion} \\
= & \mathcal{C}[C_1](l_1/x) (\mathcal{C}[C_2](l_1/x) \lambda s. \underline{Stop}) s & 3c \\
= & \mathcal{E}[x](l_1/x) (LOC? \lambda l. \mathcal{R}[x+2](l_1/x); \underline{update} l c) s & 3b \\
= & (LOC? \lambda l. \mathcal{R}[x+2](l_1/x); \underline{update} l c) l_1 s & 1b \\
= & (\mathcal{R}[x+2](l_1/x); \underline{update} l_1 c) s & \text{Def. von } LOC? \text{ und } \beta\text{-Red} \\
= & (\mathcal{E}[x+2](l_1/x); \underline{deref}; \underline{RV?}; \underline{update} l_1 c) s & 2a \\
= & (\mathcal{R}[x](l_1/x) (\lambda e_1. \mathcal{R}[2](l_1/x) \lambda e_2. \mathcal{O}[+] \langle e_1, e_2 \rangle k) s) & 1c \\
= & \mathcal{E}[x](l_1/x) (\underline{deref}; \underline{RV?} k_1) s & 2a \\
= & (\underline{deref}; \underline{RV?} k_1) l_1 s & 1b \\
= & \underline{cont} (RV? k_1) l_1 s & \text{Def. von } \underline{deref} \\
= & (RV? k_1) 1 s & \text{Def. von } \underline{cont}
\end{aligned}$$

$$\begin{aligned}
&= k_1 \ 1 \ s && \text{Def. von } RV? \\
&= \mathcal{R}[\![2]\!](l_1/x) (\underbrace{\lambda e_2. \mathcal{O}[\![+\!]\!] \langle 1, e_2 \rangle}_{k_2} k) \ s && \beta\text{-Reduktion} \\
&= \mathcal{E}[\![2]\!](l_1/x); (\underline{\text{deref}}; RV? k_2) \ s && 2a \\
&= (\underline{\text{deref}} \cdot RV? k_2) \ 2 \ s && 1a \\
&= k_2 \ 2 \ s && \text{Def. von } \underline{\text{deref}} \text{ und } RV? \\
&= \mathcal{O}[\![+\!]\!] \langle 1, 2 \rangle \ k \ s && \beta\text{-Reduktion} \\
&= k \ 3 \ s && \text{Def. von } \mathcal{O} \\
&= (\underline{\text{deref}}; RV?; \underline{\text{update}} l_1 c) \ 3 \ s \\
&= \underline{\text{update}} l_1 c \ 3 \ s && \text{Def. von } \underline{\text{deref}} \text{ und } RV? \\
&= c \ s [3/l_1] && \text{Def. von } \underline{\text{update}} \text{ und } \beta\text{-Red.} \\
&= (\mathcal{C}[\![C_2]\!](l_1/x) \ \lambda s. \underline{\text{Stop}}) \ s [3/l_1] \\
&= (\mathcal{C}[\![\text{write } x]\!](l_1/x) \ \lambda s. \underline{\text{Stop}}) (\underbrace{\langle \rangle, 3/\underline{\text{input}}, l_1}_{s_1}) \\
&= \mathcal{R}[\![x]\!](l_1/x) (\underbrace{\lambda es. (e, \underline{\text{Stop}})}_{k_3}) \ s_1 && 3f \\
&= \mathcal{E}[\![x]\!](l_1/x) (\underline{\text{deref}}; RV? k_3) \ s_1 && 2a \\
&= (\underline{\text{deref}}; RV? k_3) \ l_1 \ s_1 && 1b \\
&= \underline{\text{cont}} (RV? k_3) \ l_1 \ s_1 && \text{Def. von } \underline{\text{deref}} \\
&= (RV? k_3) \ 3 \ s_1 && \text{Def. von } \underline{\text{cont}} \\
&= k_3 \ 3 \ s_1 && \text{Def. von } RV? \\
&= \langle 3, \underline{\text{Stop}} \rangle && \beta\text{-Reduktion}
\end{aligned}$$

Bereits dieses kleine Beispiel zeigt, daß die Standardsemantik einer Sprache nicht unmittelbar dafür geeignet ist, Programm läufe zu simulieren. Die entstehenden teilausgewerteten  $\lambda$ -Ausdrücke blähen sich stark auf und wären ohne eine abkürzende Schreibweise gar nicht mehr zu verstehen. Nun liegt die Idee nahe, den

Prozeß der  $\lambda$ -Auswertungen zu automatisieren und die semantischen Gleichungen eventuell so zu transformieren, daß eine Auswertung effizienter werden kann. Ein erstes solches System wurde von Mosses in Aarhus entwickelt (s. [95] und [96]). Weitere Ansätze, aus der denotationellen Semantik, Interpreter und Compiler zu erzeugen, sollen in Kapitel 6 behandelt werden.

Die hier durchgeführte Transformation der Semantikbeschreibung von  $P$  ist natürlich auch als formaler Beweis für die Aussage: „ $P$ , angewendet auf die leere Eingabe, terminiert mit der Ausgabe  $\langle 3, \underline{\text{Stop}} \rangle$ “ zu verstehen.

## 4.5 Weitere Sprachkonzepte, analysiert im Rahmen der Standardsemantik

In diesem Abschnitt wird die Behandlung von Sprüngen und Abbrüchen diskutiert. Ferner werden einige Verallgemeinerungen des Prozedur- und Funktionskonzeptes besprochen. Schließlich wird die denotationelle Semantik einiger wichtiger Datenstrukturen wie `array`, `record` und `file` beschrieben.

### 4.5.1 Sprünge und Abbrüche

Die einfachste Form des Programmabbruchs wurde bereits mehrfach in der Semantikdefinition der Sprache  $PASCAL_0$  verwendet, und zwar im Zusammenhang mit der *Fehlerbehandlung*. Diese tritt *explizit* in den Semantikklauseln 1b und 1d auf. Dort führt die Berechnung des Wertes eines Bezeichners, der in der Umgebung ungebunden ist, bzw. die Ausführung eines Lesebefehls bei leerer Eingabe zu einer Fehlermeldung. In beiden Fällen führt das Auftreten eines Fehlers zu einem Abbruch des Programmlaufes, der semantisch durch ein Ignorieren der aktuellen Fortsetzung beschrieben wird. Eine *implizite* Fehlerbehandlung steckt auch in den Hilfsfunktionen `cont`, `update`, `ref`, `deref` und `D?`.

Eine erste Verallgemeinerung der Abbruchmöglichkeit wäre ein programmgesteueter Abbruchbefehl der Form

`if  $E$  then stop .`

Im allgemeinen möchte man jedoch, daß solche Abbrüche zu einem bestimmten Kommentar und eventuell zu einer Endberechnung führen. Dies ist in Form einer `trap` - Anweisung möglich. Eine Anweisung der Form

`trap  $C$   $I_1 : C_1, \dots, I_n : C_n$  end`

bedeutet, daß in der Anweisung  $C$  Abbrüche der Form

`escapeto  $I_\nu$`

aufreten können, deren Ausführung unmittelbar die Ausführung von  $C_\nu$  auslöst und damit die Ausführung der gesamten `trap` - Anweisung beendet.

Dazu erweitert man im Rahmen der Standardsemantik die Menge  $DV$  der bezeichneten Werte um die Anweisungsfortsetzungen  $CC$ , also

$$DV = \dots + CC + \dots ,$$

und erklärt die Semantik der neuen Anweisungen durch die semantischen Klauseln

$$\mathcal{C}[\text{trap } C I_1 : C_1, \dots, I_n : C_n \text{ end }]_{\rho c} = \mathcal{C}[C]_{\rho} [\mathcal{C}[C_1]_{\rho c} \dots \mathcal{C}[C_n]_{\rho c} / I_1 \dots I_n]_{\rho}$$

und

$$\mathcal{C}[\text{escapeto } I]_{\rho c} = \mathcal{E}[I]_{\rho}; CC? \lambda c.c .$$

Eine allgemeine Ablaufsteuerung von Programmen erlaubt die Sprunganweisung der Form

**goto**  $I$ .

Hat man die Bereichserweiterung der bezeichneten Werte wie oben vorgenommen, so lässt sich die Semantik der Sprunganweisung sofort angeben:

$$\mathcal{C}[\text{goto } I]_{\rho c} = \mathcal{E}[I]_{\rho}; CC? \lambda c.c .$$

Das Problem liegt nun darin, wie die zur Marke  $I$  gehörende Fortsetzung in der Umgebung bekannt wird. Zur Bestimmung dieser Fortsetzungen definiert man eine Funktion

$$\mathcal{J} : COM \longrightarrow ENV \longrightarrow CC \longrightarrow ENV$$

mit

$$\mathcal{J}[C]_{\rho c} = (c_1, \dots, c_n / I_1, \dots, I_n) ,$$

falls  $I_1, \dots, I_n$  genau diejenigen Marken sind, die in  $C$  vorkommen, und  $c_1, \dots, c_n$  die entsprechenden Fortsetzungen.

Nun lässt sich die Syntax von *PASCAL*<sub>0</sub> wie folgt erweitern:

$$C ::= \dots | \text{goto } I | I : C | \dots .$$

Die semantischen Klauseln zu der Funktion  $\mathcal{J}$  lauten:

#### 6. J-Semantik der Anweisungen

- (a)  $\mathcal{J}[\text{skip}]_{\rho c} = ()$
- (b)  $\mathcal{J}[E_1 := E_2]_{\rho c} = ()$
- (c)  $\mathcal{J}[C_1; C_2]_{\rho c} = \mathcal{J}[C_1]_{\rho} (\mathcal{C}[C_2]_{\rho c}) [\mathcal{J}[C_2]_{\rho c}]$
- (d)  $\mathcal{J}[\text{if } E \text{ then } C_1 \text{ else } C_2]_{\rho c} = \mathcal{J}[C_1]_{\rho c} [\mathcal{J}[C_2]_{\rho c}]$

- (e)  $\mathcal{J}[\text{while } E \text{ do } C]_{\rho c} = \mathcal{J}[C]_{\rho}(\mathcal{C}[\text{while } E \text{ do } C]_{\rho c})$
- (f)  $\mathcal{J}[\text{write } E]_{\rho c} = ()$
- (g)  $\mathcal{J}[E_1(E_2)]_{\rho c} = ()$
- (h)  $\mathcal{J}[\text{begin } D; C \text{ end }]_{\rho c} = ()$
- (i)  $\mathcal{J}[\text{goto } I]_{\rho c} = ()$
- (j)  $\mathcal{J}[I : C]_{\rho c} = \mathcal{J}[C]_{\rho c}[\mathcal{C}[C]_{\rho c}/I]$

**Bemerkungen:**

- Sprünge in Ausdrücke hinein sind nicht erlaubt (siehe 6b und 6d - 6g).
- Bei gleichen Marken innerhalb einer Sequenz von Anweisungen hat die letzte Priorität vor den vorherigen (siehe 6c).
- Kommt eine Marke innerhalb einer `if - then - else` Anweisung sowohl im `then` - Zweig als auch im `else` - Zweig vor, so wird die entsprechende Stelle im `else` - Zweig angesprungen (siehe 6d).
- Bei einem Sprung in eine `while` - Schleife wird diese Schleife ab der markierten Stelle ausgeführt (siehe 6e).
- Sprünge in innere Blöcke sind nicht erlaubt (siehe 6h).

Alle diese Festlegungen sind willkürliche Konventionen und lassen sich durch einfache Modifikationen der Klauseln von  $\mathcal{J}$  ändern. Wichtig dabei ist wie bei allen anderen Semantikdefinitionen auch, daß diese Form der Spezifikation eindeutig ist und im Gegensatz zu verbalen Verabredungen keinen Spielraum mehr für Interpretationen läßt.

Der letzte Punkt der Behandlung allgemeiner Sprunganweisungen erfordert eine entsprechende Änderung der Semantikfunktion  $\mathcal{C}$  auf der Blockanweisung und die Erweiterung von  $\mathcal{C}$  auf den neuen Anweisungsklauseln:

### 3. Semantik der Anweisungen

(a) - (g) wie oben

$$(h) \quad \mathcal{C}[\text{begin } D; C \text{ end}]_{\rho c} = \mathcal{D}[D]_{\rho} \lambda \rho'. \mathcal{C}[C]_{\rho[\rho'][\rho'']} c, \\ \text{wobei } \rho'' = \mathcal{J}[C]_{\rho[\rho'][\rho'']} c \text{ gilt.}$$

$$(i) \quad \mathcal{C}[\text{goto } I]\rho c = \mathcal{E}[I]\rho : CC? \lambda c.c$$

$$(j) \quad \mathcal{C}[I : C]\rho c = \mathcal{C}[C]\rho c$$

An dieser Definition erkennt man, daß die Gültigkeit der Marken auf den jeweils innersten umschließenden Block begrenzt ist und daß durch die rekursive Definition von  $\rho''$  die Marken auch noch in denjenigen Fortsetzungen, die an sie gebunden wurden, bekannt sind.

Anstelle der Definition von  $\rho''$  mittels einer Zusatzgleichung kann man auch unter Verwendung des fix-Operators die semantische Klausel für Blöcke wie folgt definieren:

$$\mathcal{C}[\text{begin } D; C \text{ end }]\rho c = \mathcal{D}[D]\rho \lambda \rho'. \mathcal{C}[C]\rho[\rho'][\underline{\text{fix}} \lambda \rho''. \mathcal{J}[C]\rho[\rho'][\rho''] c] c .$$

#### 4.5.2 Verallgemeinerungen des Prozedur- und Funktionskonzeptes

Die Möglichkeiten der Variation der Semantik von Prozeduren sind so vielfältig, daß sie hier nicht vollständig behandelt werden können. Es sollen jedoch fünf Modifikationen diskutiert werden:

1. Prozeduren mit mehreren Parametern
2. Rekursive Prozeduren
3. Weitere Parameterübergabemechanismen
4. Dynamische Variablenbindung
5. Funktionsprozeduren mit Zustandstransformationen

##### Prozeduren mit mehreren Parametern

Syntaktisch vereinbart man Klauseln der Form:

$$E ::= \dots | E(E_1, \dots, E_n) | \dots$$

$$C ::= \dots | E(E_1, \dots, E_n) | \dots$$

$$D ::= \dots | \text{procedure } P(I_1, \dots, I_n); C | \text{function } F(I_1, \dots, I_n); E | \dots$$

Zu jedem  $n \in IN$  ist nun je ein semantischer Bereich für Prozeduren und Funktionen mit  $n$  Parametern wie folgt erklärt:

$$\begin{aligned} PROC_n &= CC \longrightarrow EV^n \longrightarrow CC, \\ FUN_n &= EC \longrightarrow EV^n \longrightarrow CC, \text{ wobei} \\ DV &= \dots + PROC_0 + PROC_1 + \dots + FUN_0 + FUN_1 + \dots . \end{aligned}$$

Die Semantikfunktionen lassen sich nun kanonisch erweitern:

$$\begin{aligned} \mathcal{E}[E(E_1, \dots, E_n)]\rho k &= \mathcal{E}[E]\rho; FUN_n? \lambda f. \mathcal{E}[E_1]\rho \lambda e_1. \dots \mathcal{E}[E_n]\rho \lambda e_n. fk(e_1, \dots, e_n) \\ \mathcal{C}[E(E_1, \dots, E_n)]\rho c &= \mathcal{E}[E]\rho; PROC_n? \lambda p. \mathcal{E}[E_1]\rho \lambda e_1. \dots \mathcal{E}[E_n]\rho \lambda e_n. pc(e_1, \dots, e_n) \\ \mathcal{D}[\text{procedure } P(I_1, \dots, I_n); C]\rho u &= u[\lambda c(e_1, \dots, e_n). \mathcal{C}[C]\rho [e_1 \dots e_n / I_1 \dots I_n] c / P] \\ \mathcal{D}[\text{function } F(I_1, \dots, I_n); E]\rho u &= u[\lambda k(e_1, \dots, e_n). \mathcal{E}[E]\rho [e_1 \dots e_n / I_1 \dots I_n] k / F] \end{aligned}$$

### Rekursive Prozeduren

Betrachtet man die Umgebung, in der die Prozedur- und Funktionsrumpfe ausgewertet werden, so sieht man, daß dies die Deklarationsumgebung ist, die um die Bindung der formalen an die aktuellen Parameter erweitert ist. Bei rekursiven Prozeduren muß natürlich auch die Bindung des Prozedurnamens an den entsprechenden Wert aus  $PROC$  bei der Auswertung des Rumpfes bekannt sein:

$$\begin{aligned} \mathcal{D}[\text{procedure } P(I); C]\rho u &= u[p/P], \text{ wobei} \\ p &= \lambda ce. \mathcal{C}[C]\rho [pe/PI]c. \end{aligned}$$

Rekursive Funktionsprozeduren und rekursive Prozeduren mit mehreren Parametern werden analog behandelt.

### Parameterübergabemechanismen

In der Definition von  $PASCAL_0$  wurde beim Aufruf einer Prozedur der  $\mathcal{E}$ -Wert des aktuellen Parameters berechnet und an den Namen des formalen Parameters gebunden. Dieser Übergabemechanismus läßt sich zwar sehr einfach beschreiben, ist aber in den bekannten Programmiersprachen nicht vorhanden. In  $PASCAL$  gibt es zwei verschiedene Möglichkeiten:

1. Die *Wertübergabe (call by value)*.

In  $PASCAL$  besteht die Konvention, daß ein formaler Parameter, dessen

Übergabemechanismus in der Prozedurdeklaration nicht spezifiziert ist, per Wert übergeben wird. Das bedeutet, daß bei Aufruf der Prozedur der  $\mathcal{R}$ -Wert des aktuellen Parameters berechnet und in einem neuen Speicherplatz abgelegt wird, wobei während der Ausführung des Prozedurrumpfes der entsprechende formale Parameter an diesen neuen Speicherplatz gebunden ist. Ein solcher Mechanismus läßt sich auf verschiedene Weise formalisieren. Hier soll nur eine Variante angegeben werden, bei der die Deklarationssemantik nicht verändert und die Aufrufsemantik neu definiert wird:

$$\mathcal{C}[E_1(E_2)]\rho c = \mathcal{E}[E_1]\rho; \text{PROC? } \lambda p. \mathcal{R}[E_2]\rho; \underline{\text{ref}}; pc$$

Setzt man die Definition von ref ein, so wird die Wirkungsweise deutlicher:

$$\begin{aligned} \mathcal{E}[E_1]\rho; \text{PROC? } \lambda p. \mathcal{R}[E_2]\rho \lambda es. \underline{\text{new}} s &= \underline{\text{Fehler}} \longrightarrow \underline{\text{Fehler}}. \\ &\underline{\text{update}}(\underline{\text{new}} s)(pc(\underline{\text{new}} s))es. \end{aligned}$$

## 2. Die Referenzübergabe (*call by reference*).

Diese muß in der Sprache *PASCAL* durch eine Prozedurdeklaration der Form **procedure**  $P(\text{var } I); C$  explizit vereinbart werden. Bei der Referenzübergabe ist während der Ausführung von  $C$  in der Umgebung an  $I$  derjenige Speicherplatz gebunden, der bei Aufruf an den aktuellen Parameter gebunden war, d.h. insbesondere, daß der aktuelle Parameter nur an einen Speicherplatz und nicht an irgendeinen anderen bezeichnabaren Wert gebunden sein darf. Dieser Mechanismus läßt sich am einfachsten durch Abänderung der Deklarationssemantik formalisieren:

$$\mathcal{D}[\text{procedure } P(\text{var } I); C]\rho u = u \left[ \lambda c. LOC? \lambda I. \mathcal{C}[C]\rho [I/I] c/P \right].$$

Es existieren noch eine Reihe anderer Übergabemechanismen. Hier sei nur noch die aus *ALGOL 60* bekannte *Namenübergabe* (*call by name*) erwähnt, die es allerdings in *PASCAL* nicht gibt. Die Deklarationssemantik einer Vereinbarung der Form **procedure**  $P(\text{name } I); C$  ist die gleiche wie in *PASCAL*<sub>0</sub> (siehe 4c), jedoch lautet die Aufrufsemantik:

$$\mathcal{C}[E_1(E_2)]\rho c = \mathcal{E}[E_1]\rho; \text{PROC? } \lambda p. pc(\mathcal{E}[E_2]\rho) .$$

Dadurch wird bewirkt, daß bei Ausführung von  $C$  der Wert von  $E_2$  bei jedem Auftreten von  $I$  neu berechnet wird. Natürlich müssen bei solchen Definitionen Objekte der Form  $\mathcal{E}[E]\rho$ , also Elemente aus  $EC \longrightarrow CC$ , bezeichnbare und ausdrückbare Werte sein. Dann läßt sich die Berechnung eines Bezeichners  $I$  wie folgt formalisieren:

$$\mathcal{E}[I]\rho k = (\rho I = \underline{\text{ungebunden}}) \longrightarrow \underline{\text{err. }} \underline{\text{is}}_{EC \longrightarrow CC}(\rho I) \longrightarrow \rho Ik, k(\rho I) .$$

Zur Illustration soll ein kurzes Programm aufgestellt werden, dessen Semantik für die drei vorgestellten Parameterübergabemechanismen jeweils verschieden ist.

Gegeben sei folgendes *PASCAL<sub>0</sub>*-Programm:

```

P { begin
    var x = 1;
    var y = 3;
    procedure p(z);
        z := 2;
        x := 1;
        write z;
    p(if read then x else y)
end

```

Es gilt

$$\mathcal{P}[P] \langle \text{wahr}, \text{falsch} \rangle = \langle 1, \underline{\text{Stop}} \rangle .$$

Desgleichen gilt für eine Modifikation gemäß der Referenzübergabe. Vereinbart man hingegen die Wertübergabe, so ergibt sich

$$\mathcal{P}[P] \langle \text{wahr}, \text{falsch} \rangle = \langle 2, \underline{\text{Stop}} \rangle$$

und bei der Namenübergabe erhält man

$$\mathcal{P}[P] \langle \text{wahr}, \text{falsch} \rangle = \langle 3, \underline{\text{Stop}} \rangle .$$

### Dynamische Variablenbindung

In *PASCAL* und auch in *PASCAL<sub>0</sub>* wurde eine statische Variablenbindung vereinbart, d.h. Bezeichner, die in einer Prozedur frei vorkommen, werden in der Deklarationsumgebung gebunden. Dies hat den Vorteil, daß bei strukturierter Programmierung alle Größen bei der Prozedurdefinition bekannt sind. In einigen Sprachen wie z.B. *LISP* 1.5 wird wohl wegen leichter Implementierbarkeit eine dynamische Variablenbindung vorgenommen, d.h. alle frei vorkommenden Bezeichner werden in der Aufrufumgebung gebunden. Denotationell läßt sich auch die dynamische Variablenbindung recht einfach durch eine Änderung des Bereiches *PROC* der Prozedurwerte beschreiben:

$$PROC = ENV \longrightarrow CC \longrightarrow EC .$$

Damit wird es möglich, einem Prozedurwert erst eine Umgebung (die Umgebung an der Stelle des Aufrufs), dann eine Fortsetzung und schließlich den Wert des aktuellen Parameters zu übergeben. Die entsprechende Modifikation der Deklarations- und Aufrufsemantik lautet:

$$\begin{aligned} \mathcal{D}[\text{procedure } P(I); C] \rho u &= u [\lambda \rho' ce. \mathcal{C}[C] \rho' [e/I] c / P] \\ \mathcal{C}[E_1(E_2)] \rho c &= \mathcal{E}[E_1] \rho; PROC? \lambda p. \mathcal{E}[E_2] \rho; p \rho c . \end{aligned}$$

Der Nachteil der dynamischen Variablenbindung besteht darin, daß eine wichtige Eigenschaft der Programmsemantik verlorengeht: die Möglichkeit der Variablenumbenennung. Die Regel der  $\alpha$ -Konversion des  $\lambda$ -Kalküls möchte man gerne auch auf der Ebene der Programme zur Verfügung haben, um bei der Vereinbarung lokaler Variablennamen unabhängig vom Kontext zu sein. Betrachtet man folgendes Programm,

```

begin
  var x = 0;
  var y = 0;
  procedure p(z);
    write x;
  begin
    var x = 1;
    p(y)
  end ;
end

```

so erhält man bei statischer Bindung das Ergebnis  $\langle 0, \underline{Stop} \rangle$ . Hingegen ergibt sich bei dynamischer Bindung die Ausgabe  $\langle 1, \underline{Stop} \rangle$ . Ändert man in dem inneren Block die lokale Deklaration `var x = 1` in `var x' = 1` um, so führt auch die dynamische Bindung zu dem Ergebnis  $\langle 0, \underline{Stop} \rangle$ .

### Funktionsprozeduren mit Zustandstransformationen

Die Funktionsprozeduren in *PASCAL* und anderen Sprachen unterscheiden sich von denjenigen in *PASCAL*<sub>0</sub> dadurch, daß nicht nur ein Ausdruck  $E$  als Rumpf zulässig ist, sondern eine beliebige Anweisung  $C$ , in der dann der Funktionsname auf der linken Seite einer Zuweisung vorkommen kann. Die semantische Behandlung dieser allgemeinen Form

$$\text{function } F(I); C$$

ist jedoch recht kompliziert, da der Name  $F$  in  $C$  sowohl als Name für die Funktionsprozedur selbst (rekursiver Aufruf) als auch für den lokalen Speicherplatz, in dem der Funktionswert abgelegt wird (in Anweisungen der Form  $F := E$ ), steht. Um diese Schwierigkeiten zu umgehen, wird hier eine syntaktische Variante vorgestellt, die aus der Sprache *BCPL* bekannte `valof`- `resultis` Konstruktion. Man erweitert dazu die Menge der Ausdrücke um eine neue Klausel

$$E ::= \dots | \text{valof } C | \dots$$

und die Menge der Anweisungen um die Klausel

$$C ::= \dots | \text{resultis } E | \dots .$$

Intuitiv bedeutet der Ausdruck `valof C`, daß die Anweisung  $C$  solange ausgeführt wird, bis eine Anweisung der Form `resultis E` angetroffen wird. Dann liefert die Berechnung von  $E$  nach der bis dahin durchgeführten Zustandstransformation den Wert des Ausdrückes `valof C`. Wenn in  $C$  keine Anweisung der Form `resultis E` angetroffen wird, tritt ein Fehler auf. Nun läßt sich eine *PASCAL*-Funktion der Form

`function F(I);C`

in die Deklaration

`fun F(I);valof C'`

übersetzen, wobei  $C'$  aus  $C$  durch Ersetzen aller Anweisungen der Form  $F := E$  durch eine entsprechende Anweisung `resultis E` hervorgeht.

Die formale Definition der Semantik solcher Funktionsprozeduren erfordert eine neue Typisierung der Semantikfunktion  $\mathcal{C}$ , da bei der Ausführung von  $C$  die Ausdrucksfortsetzung, die bei Antreffen des Ausdrucks `valof C` vorlag, mitgeführt werden muß.

$$\begin{aligned}\mathcal{C} : COM &\longrightarrow ENV \longrightarrow EC \longrightarrow CC \longrightarrow CC \\ \mathcal{C} [\text{resultis } E] \rho k c &= \mathcal{R} [E] \rho k \\ \mathcal{E} [\text{valof } C] \rho k &= \mathcal{C} [C] \rho k \underline{\text{err}}\end{aligned}$$

Die Fehlerfortsetzung  $\underline{\text{err}}$  wird genau dann aufgerufen, wenn die Ausführung von  $C$  nicht auf eine Anweisung der Form `resultis E` stößt.

Die Semantikformeln zu allen anderen Anweisungen verwenden nur die Fortsetzung  $c$  und lassen den Parameter  $k$  unberücksichtigt.

#### 4.5.3 Datenstrukturen

In diesem Unterabschnitt soll die Sprache *PASCAL*<sub>0</sub> um einige Standarddatenstrukturen erweitert werden. Dies sind

1. Zeiger (pointer),
2. Felder (arrays),
3. Verbunde (records) und
4. Dateien (files).

Im Grunde gehören auch die elementaren Datentypen *BOOL*, *ZAHL*, *CHAR* u.s.w. zum Kapitel der Datenstrukturen. Die denotationelle Methode bietet zu ihrer Beschreibung zwei naheliegende Möglichkeiten. Entweder man bezieht die Typisierung explizit mit in die Syntax ein, oder man arbeitet syntaktisch typfrei und

integriert gewisse Typüberprüfungen in die Semantikfunktionen. Ersteres wurde in der Sprache *WHILE* exemplarisch für die Typen *BOOL* und *ZAHL* durchgeführt. In der Sprache *PASCAL*<sub>0</sub> wurde letzteres verfolgt, wobei eine konkrete Definition der Semantikfunktion  $\mathcal{O}$  mit den entsprechenden Tests dem Leser überlassen bleibt.

### Zeiger

In *PASCAL* unterscheidet man statische und dynamische Variablen. Während statische Variablen im Deklarationsteil vereinbart werden und innerhalb ihres Gültigkeitsbereiches stets dieselbe Adresse bezeichnen, werden dynamische Variablen im Anweisungsteil erzeugt und können nicht direkt durch Bezeichner angesprochen werden. Ihre Erzeugung geschieht durch Anwendung einer Funktion **new**, die im wesentlichen der Semantikfunktion **new** (siehe Abschnitt 4.3.7) entspricht und daher auch genauso bezeichnet wird. Zum Umgang mit den dynamisch erzeugten Speicherplätzen werden die *Zeigervariablen* eingeführt. Deklariert man eine solche Zeigervariable durch

**ref**  $p = E$ ,

so wird eine statische Variable  $p$  vereinbart, deren zugeordneter Speicherplatz jedoch Adressen enthalten kann, insbesondere auch Adressen von dynamisch erzeugten Variablen. Ein Aufruf der Funktion **new** mit einer Zeigervariable  $p$  als Parameter, also

**new(p)**,

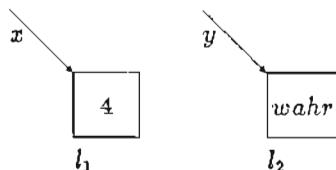
bewirkt gerade, daß die Adresse eines neuen Speicherplatzes in den an  $p$  gebundenen Speicherplatz abgelegt wird. Den Zugriff auf den Inhalt einer dynamischen Variable realisiert man wie folgt: Wenn die Zeigervariable  $p$  auf eine dynamische Variable mit der Adresse  $l$  verweist, dann bezeichnet der Ausdruck

$p \uparrow$

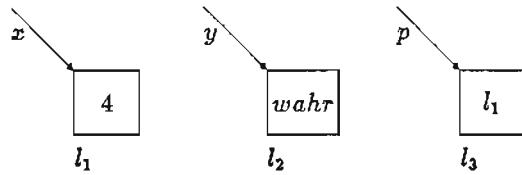
gerade den Speicherplatz mit der Adresse  $l$ .

Graphisch läßt sich dieses Konzept folgendermaßen veranschaulichen:

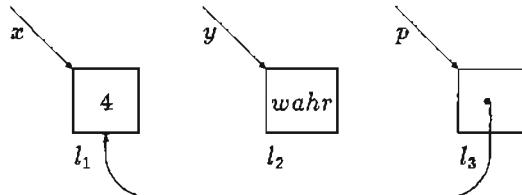
Seien  $x$  und  $y$  statische Variablen, die in einer Umgebung an die Adressen  $l_1$  bzw.  $l_2$  gebunden sind.



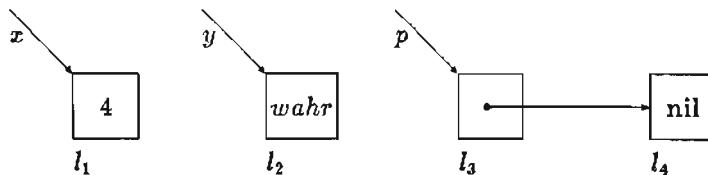
Sei **ref**  $p = x$  eine dazu lokale Deklaration.



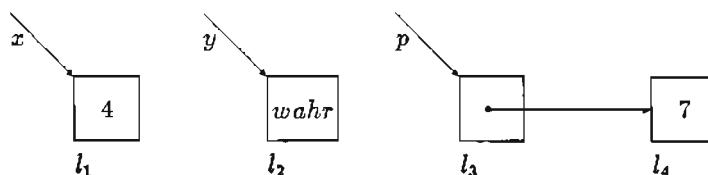
Diese Situation wird auch durch die Pfeilschreibweise dargestellt.



Eine Ausführung des Aufrufes `new(p)` legt die Adresse  $l_4$  eines neuen Speicherplatzes mit Inhalt `nil` in den von  $p$  bezeichneten Speicherplatz ab. Dabei ist `nil` der Name einer neuen Konstante.



Eine Ausführung der Anweisung  $p \uparrow := 7$  bewirkt eine entsprechende Änderung der dynamischen Variable.



Im Rahmen der Standardsemantik lässt sich eine formale Beschreibung der neuen Sprachkonstrukte leicht angeben.

$$\mathcal{D}[\text{ref } p = E] \rho u = \mathcal{L}[E] \rho; \text{ref } \lambda l. u(l/p)$$

$$\mathcal{C}[\text{new}(p)] \rho s = \text{ref } (\lambda l. c) \text{ nil } s [\text{new } s / \rho p]$$

$$\mathcal{E}[p \uparrow] \rho k = \mathcal{L}[p] \rho; \text{cont } k$$

Die Wirkungsweise dieser Definitionen soll anhand eines kurzen Programmstückes, das gerade den graphisch durchgeführten Transformationen entspricht, illustriert werden. Gegeben sei eine Umgebung  $\rho$  mit

$$\rho x = l_1 \text{ und } \rho y = l_2,$$

ein aktueller Zustand  $S$  mit

$$Sl_1 = 4 \text{ und } Sl_2 = \text{wahr}$$

und folgender innerer Block eines *PASCAL<sub>0</sub>*-Programms:

$$B \left\{ \begin{array}{l} \text{begin} \\ D \{ \text{ref } p = x; \\ C \{ C_1 \{ \text{new}(p); \\ C_2 \{ p^{\uparrow} := 7 \\ \text{end} \end{array} \right.$$

Die Semantik von  $B$ , angewendet auf die Umgebung  $\rho$ , die Fortsetzung  $\lambda s. \underline{Stop}$  und den Speicher  $S$ , lässt sich wie folgt berechnen:

$$\begin{aligned} & \mathcal{C}[B]\rho (\lambda s. \underline{Stop}) S \\ &= \mathcal{D}[\text{ref } p = x]\rho \underbrace{(\lambda \rho'. \mathcal{C}[C]\rho[\rho'] \lambda s. \underline{Stop})}_u S && 3h \\ &= \mathcal{L}[x]\rho (\underline{\text{ref}} \lambda l. u(l/p)) S && \text{s.o.} \\ &= \underline{\text{ref}} (\lambda l. u(l/p)) l_1 S && \text{Def. von } \mathcal{L} \text{ und } \mathcal{E} \\ &= \underline{\text{update}} (\underline{\text{new}} S) (u(\underline{\text{new}} S/p)) l_1 S && \text{Def. von } \underline{\text{ref}} \text{ und } \beta\text{-Red.} \\ &= \underline{\text{update}} l_3 (u(l_3/p)) l_1 S && \underline{\text{new}} S = l_3 \\ &= u(l_3/p) S [l_1/l_3] && \text{Def. von } \underline{\text{update}} \\ &= \mathcal{C}[C]\rho[l_3/p] (\lambda s. \underline{Stop}) S [l_1/l_3] && \beta\text{-Reduktion} \\ &= \mathcal{C}[\text{new}(p)]\rho[l_3/p] \underbrace{(\mathcal{C}[C_2]\rho[l_3/p] \lambda s. \underline{Stop}) S [l_1/l_3]}_c && 3c \\ &= \underline{\text{ref}} (\lambda l. c) \text{ nil } S [l_1/l_3] [\underline{\text{new}} S [l_1/l_3] / l_3] && \text{s.o.} \\ &= \underline{\text{ref}} (\lambda l. c) \text{ nil } S [l_4/l_3] && \text{Def. von } \underline{\text{new}} \\ &= \underline{\text{update}} l_4 c \text{ nil } S [l_4/l_3] && \text{Def. von } \underline{\text{ref}} \text{ und } \underline{\text{new}} \end{aligned}$$

$$\begin{aligned}
&= c \underbrace{S[l_4/l_3][\text{nil}/l_4]}_{S'} && \text{Def. von } \underline{\text{update}} \\
&= \mathcal{C}[\![p \uparrow := 7]\!] \rho[l_3/p] (\lambda s. \underline{\text{Stop}}) S' \\
&= \mathcal{E}[\![p \uparrow]\!] \rho[l_3/p] \underbrace{(\text{LOC? } \lambda l. \mathcal{R}[7] \rho[l_3/p] (\underline{\text{update}} l \lambda s. \underline{\text{Stop}}))}_{k} S' && 3b \\
&= \mathcal{L}[\![p]\!] \rho[l_3/p] (\underline{\text{cont}} k) S' && \text{s.o.} \\
&= \underline{\text{cont}} k l_3 S' && \text{Def. von } \mathcal{L} \text{ und } \mathcal{E} \\
&= k l_4 S' && \text{Def. von } \underline{\text{update}} \\
&= \mathcal{R}[\![7]\!] \rho[l_3/p] (\underline{\text{update}} l_4 \lambda s. \underline{\text{Stop}}) S' && \beta\text{-Reduktion} \\
&= \underline{\text{update}} l_4 (\lambda s. \underline{\text{Stop}}) 7 S' && 2a \text{ und } 1a \\
&= (\lambda s. \underline{\text{Stop}}) S' [7/l_4] && \text{Def. von } \underline{\text{update}} \\
&= \underline{\text{Stop}} && \beta\text{-Reduktion}
\end{aligned}$$

Der letzte erzeugte Speicher ist also

$$S[l_4/l_3][7/l_4] = [4/l_1][wahr/l_2][l_4/l_3][7/l_4],$$

wie auch in der graphischen Darstellung gezeigt.

### Felder

Das charakteristische Merkmal von Feldern ist die Zusammenfassung einer festen Anzahl *gleichartiger* Variablen unter einem Namen, wobei durch Indizierung dieses Namens auf die einzelnen Komponenten zugegriffen werden kann. Die syntaktische Spracherweiterung lässt sich wie folgt vornehmen:

$$\begin{aligned}
D &::= \dots | \mathbf{array} A[E_1..E_2] | \dots \\
E &::= \dots | E_1[E_2] | \dots
\end{aligned}$$

Hier werden der Einfachheit halber nur eindimensionale Felder behandelt, eine Erweiterung auf Mehrdimensionalität ist jedoch kanonisch.

Ein angemessener semantischer Bereich zur Modellierung der Felder besteht aus drei Komponenten.

$$\mathbf{ARRAY} = \mathbf{IN}_\perp \times \mathbf{IN}_\perp \times \mathbf{LOC}^*$$

Die erste Komponente enthält die untere Feldgrenze, die zweite die obere und die dritte alle dazugehörigen Speicherplatzadressen.

Zur Definition der Semantikfunktionen auf den neuen Konstrukten ist es nützlich, drei Hilfsfunktionen zu definieren, nämlich *news*, *newarray* und *subscript*, die die Erzeugung neuer Speicherplätze und den Zugriff auf einzelne Komponenten realisieren.

$$\underline{\text{news}} : \text{IN}_\perp \longrightarrow \text{STORE} \longrightarrow ((\text{LOC}^* \times \text{STORE}) + \{\underline{\text{Fehler}}\}_{\perp})$$

$$\begin{aligned} \underline{\text{news}} \ n \ s &= (n = 0) \longrightarrow \langle \langle \rangle, s \rangle, \\ &((\underline{\text{news}} \ (n - 1) \ s = \langle l^*, s' \rangle) \longrightarrow \\ &\quad ((\underline{\text{new}} \ s' = l) \longrightarrow \langle l^*.l, s'[\text{nil } / l] \rangle, \underline{\text{Fehler}}), \\ &\quad \underline{\text{Fehler}}) \end{aligned}$$

$$\underline{\text{newarray}} : (\text{IN}_\perp \times \text{IN}_\perp) \longrightarrow \text{EC} \longrightarrow \text{CC}$$

$$\begin{aligned} \underline{\text{newarray}} \ \langle n_1, n_2 \rangle \ k \ s &= (n_1 > n_2) \longrightarrow \underline{\text{Fehler}}, \\ &(\underline{\text{news}} \ (n_2 - n_1 + 1) \ s = \langle l^*, s' \rangle) \longrightarrow k \ \langle n_1, n_2, l^* \rangle \ s', \\ &\quad \underline{\text{Fehler}} \end{aligned}$$

$$\underline{\text{subscript}} : \text{ARRAY} \longrightarrow \text{EC} \longrightarrow \text{EC}$$

$$\begin{aligned} \underline{\text{subscript}} \ \langle n_1, n_2, l^* \rangle \ k \ e &= \text{is}_{\text{IN}_\perp} \ e \longrightarrow \\ &((n_1 \leq e \leq n_2) \longrightarrow k \ (\pi_{(e-n_1+1)} l^*), \text{err}), \\ &\quad \text{err} \end{aligned}$$

Die denotationelle Semantik der neuen Sprachkonstrukte lässt sich nun sehr einfach formalisieren.

$$\begin{aligned} \mathcal{D}[\text{array } A[E_1..E_2]]\rho u &= \\ \mathcal{R}[E_1]\rho; \text{IN}_\perp? \lambda n_1. \mathcal{R}[E_2]\rho; \text{IN}_\perp? \lambda n_2. \underline{\text{newarray}} \ \langle n_1, n_2 \rangle \ \lambda e. u(e/A) \end{aligned}$$

$$\mathcal{E}[E_1[E_2]]\rho k = \mathcal{E}[E_1]\rho; \text{ARRAY}? \lambda e. \mathcal{R}[E_2]\rho; \text{IN}_\perp? \lambda e. \underline{\text{subscript}} \ e \ k$$

Eine Verallgemeinerung dieses Feldkonzeptes durch Hinzunahme anderer Indexmengen ist leicht möglich, indem man den Bereich der Felder durch

$$\text{ARRAY} = \text{INDEX} \times \text{INDEX} \times \text{LOC}^*$$

und den Indexbereich durch

$$INDEX = IN_\perp + \overline{BOOL} + CHAR + \dots$$

definiert. Dabei müssen auch die oben verwendeten Hilfsfunktionen entsprechend geändert werden.

### Verbunde

Das charakteristische Merkmal von Verbunden ist die Zusammenfassung *verschiedenartiger* Variablen unter einem Namen, wobei auf die einzelnen Komponenten über Selektoren zugegriffen werden kann. Der Einfachheit halber soll hier wieder eine vereinfachte Version behandelt werden, bei der ein Verbund aus der Zusammenfassung endlich vieler Speicherplätze besteht, auf die mit unterschiedlichen Namen zugegriffen werden kann. Die syntaktische Spracherweiterung sieht folgendermaßen aus:

$$\begin{aligned} D &::= \dots | record R[I_1, \dots, I_n] | \dots \\ E &::= \dots | E_1.E_2 | \dots \end{aligned}$$

In *PASCAL* besteht die Möglichkeit , eine Anweisung  $C$ , die zur Manipulation eines Verbundes  $R$  dient, mit dem Präfix „with  $R$  do“ zu versehen, was bewirkt, daß in  $C$  auf die Komponenten von  $R$  direkt mit den entsprechenden Selektornamen zugegriffen werden kann, ohne jedesmal den Namen  $R$  als Präfix zu verwenden. Zu diesem Zweck erweitert man die syntaktischen Klauseln zu

$$C ::= \dots | with E do C | \dots .$$

Eine zweckmäßige Bereichsdefinition für Verbunde ist es, den Bereich der Umgebungen zu übernehmen, da ja auch bei Verbunden eine Zuordnung von Bezeichnern an bezeichnbare Werte vorgenommen wird.

$$RECORD = ID \longrightarrow (DV + \{\underline{ungebunden}\}_\perp) = ENV$$

Die denotationelle Semantik der neuen Sprachkonstrukte erhält man durch folgende Gleichungen:

$$\begin{aligned} \mathcal{D}[\![\text{record } R[I_1, \dots, I_n]]\!]_{\rho u s} &= \\ (\underline{news} \ ns = \langle \langle l_1, \dots, l_n \rangle, s' \rangle) &\longrightarrow u((l_1 \dots l_n / I_1 \dots I_n) / R) \ s', \\ \underline{Fehler} & \end{aligned}$$

$$\begin{aligned} \mathcal{E}[\![E_1.E_2]\!]_{\rho k} &= \mathcal{E}[\![E_1]\!]_{\rho}; \ RECORD? \lambda \rho'. \mathcal{E}[\![E_2]\!]_{\rho[\rho']} \ k \\ \mathcal{C}[\![\text{with } E \text{ do } C]\!]_{\rho c} &= \mathcal{E}[\![E]\!]_{\rho}; \ RECORD? \lambda \rho'. \mathcal{C}[\![C]\!]_{\rho[\rho']} \ c . \end{aligned}$$

### Dateien

Die einfachste Form der Strukturierung von Daten in *PASCAL* ist die *Sequenz*. Wie allgemein üblich, bezeichnet man eine linear geordnete Menge von Daten mit einem Zeiger auf eine aktuelle Komponente als *Datei* (*file*). Der Zugriff auf Dateikomponenten geschieht über eine Puffervariable, deren Inhalt auf die aktuelle Komponente geschrieben bzw. mit dem Wert der aktuellen Komponente überschrieben werden kann. Zusätzlich benötigt man bestimmte Befehle zum Verschieben des Zeigers. Dies ist i.a. nur beschränkt möglich. In *PASCAL* bewirkt der Befehl

**reset *F***

ein Setzen dieses Zeigers auf den Anfang der Datei *F* und eine Aktualisierung der Puffervariable  $F^\uparrow$  mit dem Inhalt der ersten Dateikomponente, der Befehl

**rewrite *F***

ein Löschen der Datei *F* und der Befehl

**get *F***

das Verschieben des Zeigers um eine Stelle nach rechts und eine entsprechende Aktualisierung der Puffervariable. Jede *PASCAL*-Datei endet mit einem speziellen Wert, der EOF-Marke, sodaß ein Test

**eof *F***

auf das Ende der Datei *F* möglich ist. Schließlich bewirkt der Befehl

**put *F*,**

daß der Inhalt der Puffervariable  $F^\uparrow$  auf die aktuelle Komponente der Datei *F* geschrieben und der Zeiger um eine Stelle nach rechts verschoben wird.

Syntaktisch gibt es also folgende Spracherweiterungen:

$$\begin{aligned} D &::= \dots | \text{file } F | \dots \\ E &::= \dots | \text{eof } E | \dots \\ E &::= \dots | \text{reset } E | \text{rewrite } E | \text{get } E | \text{put } E | \dots . \end{aligned}$$

Der semantische Bereich der Dateien läßt sich wie folgt definieren:

$$FILE = RV^* \times IN_\perp \times LOC ,$$

wobei die erste Komponente alle in der Datei eingetragenen Datenelemente enthält, die zweite Komponente die Position des Zeigers bestimmt und in der dritten Komponente die Adresse der Puffervariable enthalten ist.

Die semantischen Funktionen lassen sich wieder nach Einführung geeigneter Hilfsfunktionen eleganter aufschreiben. Da der Bereich *FILE* nur die Adresse der Puffervariable enthält, man sich aber in der Regel für deren Inhalt interessiert, lassen

sich die Hilfsfunktionen vereinfachen, wenn ein zusätzlicher Bereich der Dateizustände verwendet wird.

$$\text{FILESTATE} = \text{RV}^* \times \text{IN}_\perp \times (\text{RV} + \{\text{frei}\}_\perp)$$

In Analogie zu den oben erläuterten Dateioperationen definiert man vier Hilfsfunktionen resetf, rewritef, getf und putf, die Dateizustände manipulieren, d.h. folgenden Typs sind:

$$\text{FILESTATE} \longrightarrow (\text{FILESTATE} + \{\text{Fehler}\}_\perp).$$

$$\underline{\text{resetf}} \langle e^*, n, e \rangle = \underline{\text{null}} \ e^* \longrightarrow \langle e^*, 1, \underline{\text{frei}} \rangle, \langle e^*, 1, \pi_1 \ e^* \rangle$$

$$\underline{\text{rewritef}} \langle e^*, n, e \rangle = \langle \langle \rangle, 1, \underline{\text{frei}} \rangle$$

$$\begin{aligned} \underline{\text{getf}} \langle e^*, n, e \rangle &- (n > \underline{\text{length}} \ e^*) \longrightarrow \underline{\text{Fehler}}, \\ &(n = \underline{\text{length}} \ e^*) \longrightarrow \langle e^*, n+1, \underline{\text{frei}} \rangle, \\ &(n < \underline{\text{length}} \ e^*) \longrightarrow \langle e^*, n+1, \pi_{n+1} \ e^* \rangle \end{aligned}$$

$$\underline{\text{putf}} \langle e^*, n, e \rangle = (n = \underline{\text{length}} \ e^* + 1) \longrightarrow \langle e^*.e, n+1, \underline{\text{frei}} \rangle, \underline{\text{Fehler}}$$

Zur Einbettung dieser direkten Hilfsfunktionen in die Semantikfunktionen erzeugt man aus ihnen Fortsetzungstransformationen mittels einer letzten Hilfsfunktion do.

$$\underline{\text{do}} : \text{FILESTATE} \longrightarrow (\text{FILESTATE} + \{\text{Fehler}\}_\perp) \longrightarrow CC \longrightarrow EC$$

$$\begin{aligned} \text{do } f \ c \ l \ s &- \underline{\text{isLOC}} \ l \longrightarrow ((\underline{\text{si}} = \langle e^*, n, l' \rangle) \longrightarrow \\ &((\underline{\text{sl}} = e) \longrightarrow ((f \langle e^*, n, e \rangle = \langle d^*, n', e' \rangle) \longrightarrow \\ &c \ s[(d^*, n', e) \ e'/l \ l'], \underline{\text{Fehler}}), \underline{\text{Fehler}}), \underline{\text{Fehler}}. \end{aligned}$$

Damit ist die denotationelle Beschreibung der Dateimanipulationen sehr einfach.

$$\begin{aligned} \mathcal{C}[\text{reset } E] \rho c &= \mathcal{E}[E] \rho; \text{do } \underline{\text{resetf}} \ c \\ \mathcal{C}[\text{rewrite } E] \rho c &= \mathcal{E}[E] \rho; \text{do } \underline{\text{rewritef}} \ c \\ \mathcal{C}[\text{get } E] \rho c &= \mathcal{E}[E] \rho; \text{do } \underline{\text{getf}} \ c \\ \mathcal{C}[\text{put } E] \rho c &= \mathcal{E}[E] \rho; \text{do } \underline{\text{putf}} \ c \end{aligned}$$

Eine Formalisierung der Deklaration von Dateien sowie des Tests auf Leerheit lässt sich ebenfalls recht einfach aufschreiben.

$$\begin{aligned} \mathcal{D}[\text{file } F] \rho us &= \\ (\underline{\text{news}} \ 2 \ s = \langle \langle l_1, l_2 \rangle, s' \rangle) &\longrightarrow u(l_1, l_2/F, F^\uparrow) \ s'[(\langle \langle \rangle, 1, l_2 \rangle / l_1], \underline{\text{Fehler}} \\ \mathcal{E}[\text{eof } E] \rho k &= \\ \mathcal{E}[E] \rho; \text{LOC? } \lambda ls.(\underline{\text{sl}} = \langle e^*, n, l' \rangle) &\longrightarrow k(n > \underline{\text{length}} \ e^*)s, \underline{\text{Fehler}} \end{aligned}$$

Die in der Sprache *PASCAL* zusätzlich erlaubten Befehle

read ( $F, x$ ) und write ( $F, x$ )

können durch die Anweisungsfolgen

$x := F \uparrow ; \text{get } F$  bzw.  $F \uparrow := x ; \text{put } F$

definiert werden, und daher lässt sich ihre Semantik direkt durch die Semantik der entsprechenden Anweisungsfolge definieren.

## Kapitel 5

# Funktionale Programmiersprachen

Im vorherigen Kapitel diente der  $\lambda$ -Kalkül, angereichert um einige Basisfunktionen und -prädikate, als Metasprache zur Formalisierung der Semantik imperativer Programmiersprachen. Der gleiche Kalkül, sogar mit weniger Basisfunktionen, eignet sich auch sehr gut, ein gewünschtes Ein-/Ausgabeverhalten *direkt* als Funktion zu beschreiben. Daher liegt es nahe, Programmiersprachen zu betrachten, deren Sprachelemente die des  $\lambda$ -Kalküls oder der verwandten kombinatorischen Logik sind. Bei solchen Sprachen ist die Metasprache der imperativen Sprachen zur Objektsprache geworden, und damit besteht die Spezifikation der Semantik nur noch aus der Angabe der semantischen Bereiche und der Interpretation der Funktions- und Prädikatssymbole. Ein funktionales Programm besteht dann aus einem Ausdruck  $A$ , der selbst das von ihm definierte Objekt bezeichnet, d.h. die Semantikfunktion wird durch die Gleichung

$$[A] = A$$

bestimmt. Damit lassen sich Aussagen über Programmeigenschaften wesentlich kürzer beweisen.

Die am meisten verbreitete Sprache dieser Art ist LISP (siehe [86] und [87]), wobei leider aus Effizienzgründen die Semantik von LISP von der des  $\lambda$ -Kalküls abweicht. Daher muß eine denotationelle Spezifikation das entsprechende Interpreterverhalten simulieren, wodurch auch Beweise über Programmeigenschaften wieder komplexer werden.

In Abschnitt 5.1 wird die Sprache Kern-LISP vorgestellt und ihre Semantik denotationell spezifiziert. Anschließend wird die Abweichung gegenüber der Semantik des  $\lambda$ -Kalküls charakterisiert und gezeigt, daß in Kern-LISP die Gesetze des  $\lambda$ -Kalküls nicht gelten (siehe auch Eick und Fehr [33]).

Eine weitere funktionale Sprache, die jedoch mehr an der kombinatorischen Logik orientiert ist, wird durch die FP-Systeme von Backus [9] definiert, die in Abschnitt

5.2 behandelt werden. Die Semantik dieser Sprache ist konsistent mit der mathematischen Theorie der Kombinatoren. Die Programmiertechnik in FP-Systemen ist jedoch dadurch erschwert, daß die Objekte, auf denen operiert werden soll, nicht bezeichnet werden können.

Schließlich wird in Abschnitt 5.3 als Repräsentant einer Klasse von modernen funktionalen Sprachen, die sowohl die  $\lambda$ -Kalkül-Semantik verwenden als auch leicht lesbare Funktionsdefinitionen erlauben, die funktionale Sprache Miranda von Turner [126] vorgestellt.

Da das Thema funktionale Programmierung im Rahmen einer Semantikvorlesung nicht umfassend behandelt werden kann, sei an dieser Stelle auf das Lehrbuch von Lippe und Simon [75] verwiesen. Daneben bietet das Buch von Henderson [44] eine praktische Einführung in Programmiertechniken im funktionalen Stil.

## 5.1 Die Programmiersprache LISP

Die Grundidee bei der Entwicklung von LISP war die Einsicht, daß der  $\lambda$ -Kalkül, angereichert um Listenoperationen, eine mächtige und mathematisch formalisierte Sprache ist, die sich besonders gut dazu eignet, komplexe Listenoperationen strukturiert zu programmieren. Eine naheliegende Möglichkeit der maschinellen Auswertung von  $\lambda$ -Ausdrücken ist die Implementierung der  $\alpha$ -,  $\beta$ -,  $\delta$ - und  $\eta$ -Konversion sowie die Verwendung einer korrekten Auswertestrategie. Leider stellt sich heraus, daß dies aus folgenden Gründen nicht sehr effizient möglich ist:

1. Die Suche nach freien Variablen im Argument, die im Rumpf gebunden sind, die Erzeugung neuer Variablennamen sowie die entsprechenden Umbenennungen sind sehr aufwendig (vgl. Definition 3.26.1 und 2).
2. Die Substitution von Argumenten für die entsprechenden formalen Parameter ist ineffizient, da im allgemeinen große Ausdrücke mehrfach kopiert werden müssen.

Also wurde beim Entwurf des Interpreters für LISP die Idee verfolgt, einen Kellerspeicher (a-list für association-list) zur Ablage von Paaren der Form

$$[\text{formaler Parameter} \mid \text{aktueller Parameter}]$$

zu verwenden, so daß erst ersetzt wird, wenn ein formaler Parameter bei der Auswertung des Ausdruckes gebraucht wird. Diese Methode ist jedoch im allgemeinen nicht korrekt, wie folgendes Beispiel zeigt:

Seien  $a$  und  $b$  Konstanten. Der  $\lambda$ -Ausdruck

$$t = (\lambda x.(\lambda y.(\lambda x.y)b)x)a$$

läßt sich wie folgt reduzieren:

$$\begin{aligned} t &\xrightarrow{\beta} (\lambda y.(\lambda x.y)b)a \\ &\xrightarrow{\beta} (\lambda x.a)b \\ &\xrightarrow{\beta} a \end{aligned}$$

Der Wert von  $t$  bzgl. der Semantik des  $\lambda$ -Kalküls ist also die Konstante  $a$ . Eine einfache Kellerimplementierung arbeitet wie folgt:

Ausdruck	Keller
$(\lambda x.(\lambda y.(\lambda x.y)b)x)a$	$<>$
$(\lambda y.(\lambda x.y)b)x$	$<[x   a]>$
$(\lambda x.y)b$	$<[y   x], [x   a]>$
$y$	$<[x   b], [y   x], [x   a]>$
$b$	$<[x   b], [y   x], [x   a]>$

Hier sieht man bereits, daß die Kellerimplementierung ein falsches Ergebnis liefert, da der Wert von  $y$  auf den Wert von  $x$  zurückgeführt wird und dieser dann dynamisch auf den Wert  $b$  gesetzt wird, anstatt die statische Bindung von  $x$  an  $a$  zu erkennen.

Ein Verbot, nach Variablen innerhalb ihres Gültigkeitsbereiches erneut zu abstrahieren, würde erstens die Modularität zerstören und zweitens auch nur für eine eingeschränkte Ausdrucksmenge eine korrekte Kellerimplementierung gestatten. Dazu betrachte man folgendes Beispiel:

$$\begin{aligned} (\lambda z.zz)\lambda x.\lambda y.xy &\xrightarrow{\beta} (\lambda x.\lambda y.xy)\lambda x.\lambda y.xy \\ &\xrightarrow{\beta} \lambda y.(\lambda x.\lambda y.xy)y \end{aligned}$$

Obwohl in diesem Ausdruck zunächst nach keiner Variable mehrfach abstrahiert wird, ist dies nach zwei  $\beta$ -Reduktionen bereits der Fall.

Beim Entwurf der Sprache LISP glaubte man, daß die Einschränkung der Auswertungsstrategie auf ‘call-by-value’ das Problem lösen würde. ‘Call-by-value’ bedeutet, daß ein aktueller Parameter zuerst ausgewertet wird, bevor der dazugehörige formale Parameter an ihn gebunden wird. Obwohl diese Strategie nicht vollständig ist, kann man im allgemeinen Ausdrücke so formulieren, daß diese Strategie zum Ergebnis führt (siehe Plotkin [102]).

In unserem ersten Beispiel hilft dieses Vorgehen:

Ausdruck	Keller
$(\lambda x.(\lambda y.(\lambda x.y)b)x)a$	$<>$
$(\lambda y.(\lambda x.y)b)x$	$<[x   a]>$
$(\lambda x.y)b$	$<[y   a], [x   a]>$
$y$	$<[x   b], [y   a], [x   a]>$
$a$	$<[x   b], [y   a], [x   a]>$

Leider ist dies aber immer noch keine korrekte Implementierung, wie man am zweiten Beispiel erkennen kann:

Ausdruck	Keller
$(\lambda z.zz)\lambda x.\lambda y.xy$	$<>$
$zz$	$<[z   \lambda x.\lambda y.xy]>$
$(\lambda x.\lambda y.xy)z$	$<[z   \lambda x.\lambda y.xy]>$
$\lambda y.xy$	$<[x   \lambda x.\lambda y.xy], <[z   \lambda x.\lambda y.xy]>$
$\lambda y.(\lambda x.\lambda y.xy)y$	$<[x   \lambda x.\lambda y.xy], <[z   \lambda x.\lambda y.xy]>$
$\lambda y.\lambda y.xy$	$<[x   y], [x   \lambda x.\lambda y.xy], <[z   \lambda x.\lambda y.xy]>$
$\lambda y.\lambda y.yz$	$<[x   y], [x   \lambda x.\lambda y.xy], <[z   \lambda x.\lambda y.xy]>$

Das Problem bei dieser Auswertung besteht darin, daß der Gesamtausdruck nicht vom Basistyp ist, d.h. es bleiben äußere Abstraktionen bestehen, zu denen kein Argument existiert. Das hat zur Folge, daß auch bei einer ‘call-by-value’-Strategie aktuelle Parameter selbst nach ihrer Auswertung nicht notwendigerweise geschlossene Ausdrücke sind, so daß durch ihre Substitution falsche Bindungen entstehen können.

Ein weiteres Problem ergibt sich, wenn aktuelle Parameter nicht vom Basistyp sind. Da sich funktionale Ausdrücke nicht auf Konstanten reduzieren lassen, können ebenfalls falsche Variablenbindungen durch die Substitution von aktuellen Parametern entstehen. Für später entwickelte Dialekte von LISP hat man daher eine spezielle Behandlung funktionaler Argumente implementiert, auf die hier nicht weiter eingegangen werden soll (vgl. Allen [2]).

Der Kern der Sprache LISP, der auch als Metasprache zur Definition des Interpreters verwendet wird, beschränkt sich auf Ausdrücke vom Basistyp, die auch nur Parameter vom Basistyp zulassen. Diese Sprache werden wir im folgenden formal definieren und nachweisen, daß gegen alle Absicht die Semantik dennoch nicht mit den Gesetzen des  $\lambda$ -Kalküls verträglich ist.

### 5.1.1 Syntax von Kern-LISP

#### Syntaktische Bereiche mit dazugehörigen Metavariablen

##### Primitive Bereiche

$X$  Bereich der Variablen  $x, y, z, f, g$

$N$  Bereich der Zahlen  $n, m$

$W$  Bereich der Worte  $v, w$

$B$  Bereich der Wahrheitswerte  $b$

$B := \{\underline{t}, \underline{f}\}$

$O$  Bereich der Basisoperationen  $o$

$O := \{\underline{car}^{(1)}, \underline{cdr}^{(1)}, \underline{cons}^{(2)}, \underline{eq}^{(2)}, \underline{atom}^{(1)}, \underline{null}^{(1)}, +^{(2)}, *^{(2)}, -^{(2)}, \dots\}$

##### Zusammengesetzte Bereiche

$A$  Bereich der Atome  $a$

$L$  Bereich der Listen  $l$

$E$  Bereich der Ausdrücke  $e$

$F$  Bereich der Funktionen  $fn$

#### Syntaktische Klauseln

$a ::= w \mid b \mid n$

$l ::= \text{err} \mid a \mid (l^*) \mid (l_1.l_2)$

$e ::= l \mid x \mid fn[e_1; \dots; e_n] \mid [e_1, \dots; e_{n_1} \rightarrow e_{n_2}]$

$fn ::= o \mid f \mid \lambda[x_1; \dots; x_n]; e \mid \text{label} [f; fn]$

##### Zunächst einige Erläuterungen zur Syntax:

In Anlehnung an die LISP-Konvention wurden kleine Buchstaben als Metavariablen gewählt.

Die primitiven Bereiche  $X, N$  und  $W$  sind hier nicht vollständig definiert, da dies systemabhängig ist und auf die Semantikspezifikation keinen Einfluß hat.

Auch die Menge der Basisoperationen variiert je nach Implementierung, wobei die hier explizit genannten stets vorhanden sind. Der hochgestellte Index bezeichnet die Stelligkeit einer Basisoperation. In LISP sind alle Operationen über dem Bereich  $L$  der Listen erklärt, dadurch ist ihr Typ durch Angabe der Stelligkeit eindeutig bestimmt.

Wie bereits aus der Syntaxdefinition implizit hervorgeht, sind in diesem Bereich  $L$

recht allgemeine Listenstrukturen zusammengefaßt. Die entsprechende Bereichsgleichung lautet:

$$L = A + L^* + (L \times L) .$$

In LISP wird im allgemeinen nur die Gleichung

$$L = A + (L \times L)$$

verwendet, wobei Objekte aus  $L^*$  auf Objekte aus  $A + (L \times L)$  durch folgende Injektion  $i$  abgebildet werden:

$$\begin{aligned} i() &= \underline{\text{nil}} \quad (\text{ein spezielles Wort aus } A) \\ i l_1.l &= \langle l_1, l \rangle \quad (\text{meist als 'dotted pair' } (l_1.l) \text{ geschrieben}). \end{aligned}$$

Syntaktisch ist es jedoch eine brauchbare Spracherweiterung, wenn sowohl die Listen- als auch die Paarschreibweise erlaubt ist.

Vergleicht man dies mit der Syntax des getypten  $\lambda$ -Kalküls, so stellt man fest, daß sich Kern-LISP davon nur geringfügig unterscheidet:

1. Die Verzweigung ist ein eigenes syntaktisches Konstrukt und nicht Element der Menge der Basisoperationen.
2. Die Typen der LISP-Objekte sind auf den Basistyp  $L$  und funktionale Typen erster Stufe der Form  $L^n \rightarrow L$  eingeschränkt.
3. Wegen der unter Punkt 2. genannten Typeinschränkung muß die Rekursion explizit (als Sprachelement `label`) aufgenommen werden, da der Operator `fix` zur Definition rekursiver Listenfunktionen vom Typ

$$[L^n \rightarrow L] \rightarrow [L^n \rightarrow L]$$

ist.

4. Die Applikation und Abstraktion sind n-stellig anstatt monadisch wie im  $\lambda$ -Kalkül.
5. Die Bezeichnungen der Basisobjekte und -operationen weichen geringfügig ab. Listenklammern sind rund anstatt spitz, Applikationsklammern sind eckig anstatt rund, und die Basisoperationen haben folgende Entsprechungen:

$$\begin{aligned} \text{car} &\hat{=} \underline{\text{hd}} \\ \text{cdr} &\hat{=} \underline{\text{tl}} \\ \text{cons}[k;l] &\hat{=} k.l \\ \text{atom} &\hat{=} \underline{\text{is}_A} \end{aligned}$$

Mit diesen Erläuterungen kann man unmittelbar die Semantik der Sprache Kern-LISP als die Semantik des  $\lambda$ -Kalküls verstehen. Der Vollständigkeit halber sei diese hier noch einmal formal definiert, wobei erneut darauf hingewiesen wird, daß es sich dabei *nicht* um die Semantik von LISP handelt, wie sie in [87] durch den Interpreter `evalquote` von McCarthy et al. definiert ist, sondern um eine denotationelle Semantikbeschreibung von Kern-LISP mit *statischer* Variablenbindung.

### 5.1.2 Statische Semantik von Kern-LISP

#### Semantische Bereiche mit dazugehörigen Metavariablen

Die semantischen Bereiche für Atome und Listen seien die gleichen wie die syntaktischen.

$$U = X \longrightarrow L + [L^* \longrightarrow L] \text{ Bereich der Umgebungen } \sigma$$

#### Semantikfunktionen

$$\begin{aligned} \mathcal{E} &: E \longrightarrow U \longrightarrow L \\ \mathcal{F} &: F \longrightarrow U \longrightarrow L^* \longrightarrow L \end{aligned}$$

#### Semantische Klauseln

##### 1. Semantik der Ausdrücke

$$(a) \quad \mathcal{E}[l]\sigma = l$$

$$(b) \quad \mathcal{E}[x]\sigma = \sigma x$$

$$(c) \quad \mathcal{E}[fn[e_1; \dots; e_n]]\sigma = \mathcal{F}[fn]\sigma(\mathcal{E}[e_1]\sigma, \dots, \mathcal{E}[e_n]\sigma)$$

$$(d) \quad \mathcal{E}[[e_{1_1} \longrightarrow e_{1_2}; \dots; e_{n_1} \longrightarrow e_{n_2}]]\sigma = \mathcal{E}[e_{1_1}]\sigma \longrightarrow \mathcal{E}[e_{1_2}]\sigma,$$

...

$$\mathcal{E}[e_{n_1}]\sigma \longrightarrow \mathcal{E}[e_{n_2}]\sigma, \underline{ff}$$

##### 2. Semantik der Funktionen

$$(a) \quad \mathcal{F}[o]\sigma = o$$

$$(b) \quad \mathcal{F}[f]\sigma = \sigma f$$

$$(c) \quad \mathcal{F}[\lambda[[x_1; \dots; x_n]; e]]\sigma = \lambda l_1 \dots l_n. \mathcal{E}[e]\sigma[l_1 \dots l_n/x_1 \dots x_n]$$

$$(d) \quad \mathcal{F}[\text{label } [f; fn]]\sigma = \underline{fix} \lambda F. \mathcal{F}[fn]\sigma[F/f]$$

Für die Sprache Kern-LISP zusammen mit der statischen Semantik gilt nun die gewünschte Gleichheit von Programmiersprache und Metasprache, denn für geschlossene Ausdrücke und Funktionen gilt bis auf unterschiedliche Konventionen in der Schreibweise:

$$[e] = e \text{ und } [fn] = fn.$$

### 5.1.3 Operationelle Semantik von Kern-LISP

Im LISP 1.5 Programmer's Manual von McCarthy et al. [87] wird Kern-LISP als Metasprache zur Definition der Semantik von LISP verwendet. Eine universelle Funktion *evalquote* zur Auswertung von LISP-Funktionen wird dort in Kern-LISP definiert. Um den Vergleich zur statischen Semantik zu ermöglichen, soll der Interpreter *evalquote* hier kurz vorgestellt werden. Da alle LISP-Funktionen auf Listen arbeiten, müssen Ausdrücke und Funktionen als Listen dargestellt werden, um sie der Auswertungsfunktion *evalquote* als Argument übergeben zu können. Dazu wird eine Übersetzungsfunktion  $\tau$  definiert.

Zunächst erweitert man die Menge  $A$  der Atome um die Menge  $X$  der Variablen, um die Menge  $O$  der Basisoperationssymbole und um die Elemente *err*, *quote*, *cond*, *lambda* und *label*.

#### Definition 5.1

$$\tau : (E + F) \longrightarrow L$$

$$\begin{aligned} \tau \text{ err} &= \underline{\text{err}} \\ \tau a &= (\underline{\text{quote}} a) \\ \tau (l_1 \dots l_n) &= (\tau l_1 \dots \tau l_n) \\ \tau (l_1.l_2) &= (\tau l_1 . \tau l_2) \\ \tau x &= x \\ \tau fn[e_1; \dots; e_n] &= (\tau fn \; \tau e_1 \dots \tau e_n) \\ \tau [e_{1_1} \longrightarrow e_{1_2}; \dots; e_{n_1} \longrightarrow e_{n_2}] &= (\underline{\text{cond}}(\tau e_{1_1} \; \tau e_{1_2}) \dots (\tau e_{n_1} \; \tau e_{n_2})) \\ \tau o &= o \\ \tau f &= f \\ \tau \lambda[x_1; \dots; x_n]; e &= (\underline{\text{lambda}} (x_1 \dots x_n) \; \tau e) \\ \tau \text{label } [f; fn] &= (\underline{\text{label}} f \; \tau fn) \end{aligned}$$

■

Nun soll die Interpretersfunktion *evalquote* in Kern-LISP so definiert werden, daß gilt:

$$\underline{\text{evalquote}}[\tau fn; (e_1 \dots e_n)] = fn[e_1; \dots; e_n].$$

Wie zu Beginn dieses Abschnittes erläutert, ist die Idee dieses Interpreters, einen Kellerspeicher zur Ablage von Variablenbindungen zu verwenden. Dieser Kellerspeicher wird 'association-list' oder kurz 'a-list' genannt.

Zur Manipulation der a-list werden zwei Hilfsfunktionen definiert:

$$\begin{aligned} \underline{\text{pairlist}}[x; y; a] &= [\underline{\text{null}}[x] \longrightarrow a; \\ &\quad \underline{t} \longrightarrow \underline{\text{cons}}[\underline{\text{cons}}[\underline{\text{car}}[x]; \underline{\text{car}}[y]]; \underline{\text{pairlist}}[\underline{\text{cdr}}[x]; \underline{\text{cdr}}[y]; a]]] \\ \underline{\text{assoc}}[x; a] &= [\underline{\text{eq}}[\underline{\text{caar}}[a]; x] \longrightarrow \underline{\text{car}}[a]; \\ &\quad \underline{t} \longrightarrow \underline{\text{assoc}}[x; \underline{\text{cdr}}[a]]] . \end{aligned}$$

Wie man sieht, wird auch in LISP die Gleichungsschreibweise der Notation unter Verwendung von `label` bevorzugt. Eine weitere syntaktische Vereinfachung ist die Abkürzung von Kompositionen aus `car` und `cdr` durch die entsprechende Folge von `a`'s bzw. `d`'s zwischen den Buchstaben `c` und `r`.

**Beispiel:**

$$\underline{\text{cddar}}[x] = \underline{\text{cdr}}[\underline{\text{cdr}}[\underline{\text{car}}[x]]] .$$

Die Wirkungsweise von `pairlist` entspricht der Modifikation von Umgebungen in der denotationellen Semantik. Jede a-list beschreibt eindeutig eine Umgebung, und in der vertrauten Notation könnte man sagen:

$$\underline{\text{pairlist}}[x; y; a] \doteq a[x/y] .$$

**Beispiel:**

$$\begin{aligned} \underline{\text{pairlist}}[(a\ b\ c); (u\ v\ w); ((d.x)(e.y))] &= \underline{\text{cons}}[\underline{\text{cons}}[a; u]; \underline{\text{pairlist}}[(b\ c); (v\ w); ((d.x)(e.y))]] \\ &= ((a.u).\underline{\text{cons}}[\underline{\text{cons}}[b; v]; \underline{\text{pairlist}}[(c); (w); ((d.x)(e.y))]]) \\ &= ((a.u).((b.v).\underline{\text{cons}}[\underline{\text{cons}}[c; w]; \underline{\text{pairlist}}[(); (); ((d.x)(e.y))]])) \\ &= ((a.u).((b.v).((c.w).((d.x)(e.y))))) \\ &= ((a.u)(b.v)(c.w)(d.x)(e.y)) \end{aligned}$$

Die Hilfsfunktion `assoc` bestimmt zu einer Variablen `x` und einer a-list `a` das entsprechende Tupel

$$(x.\text{`Wert von } x \text{ bzgl. } a') .$$

**Beispiel:**

$$\begin{aligned} \underline{\text{assoc}}[y; ((a.(m\ n))(y.(car\ x))(c.(\underline{\text{quote}}\ m))(c.(cdr\ x)))] &= \underline{\text{assoc}}[y; ((y.(\underline{\text{car}}\ x))(c.(\underline{\text{quote}}\ m))(c.(cdr\ x)))] \\ &= (y.(\underline{\text{car}}\ x)) . \end{aligned}$$

Die universelle Funktion *evalquote* verwendet zur Berechnung eines Funktionswertes im wesentlichen die Hilfsfunktionen *eval* und *apply*. *apply* bestimmt den Wert einer LISP-Funktion, angewendet auf eine Argumentliste, bzgl. einer a-list, und *eval* bestimmt den Wert eines Ausdruckes bzgl. einer a-list.

**Definition 5.2 (Der Kern-LISP-Interpreter *evalquote*)**

*evalquote*[*fn*; *x*] = *apply*[*fn*; *x*; *nil*]

*apply*[*fn*; *x*; *a*] =

[*atom*[*fn*] → [ *eq*[*fn*; *car*] → *caar*[*x*];  
*eq*[*fn*; *cdr*] → *cadr*[*x*];  
*eq*[*fn*; *cons*] → *cons*[*car*[*x*]; *cadr*[*x*]];  
*eq*[*fn*; *atom*] → *atom*[*car*[*x*]];  
*eq*[*fn*; *eq*] → *eq*[*car*[*x*]; *cadr*[*x*]];  
*tt* → *apply*[*eval*[*fn*; *a*]; *x*; *a*]];

*eq*[*car*[*fn*]; *lambda*] → *eval*[*caddr*[*fn*]; *pairlist*[*cadr*[*fn*]; *x*; *a*]];

*eq*[*car*[*fn*]; *label*] → *apply*[*caddr*[*fn*]; *x*; *cons*[*cons*[*cadr*[*fn*]]; *caddr*[*fn*]]; *a*]]

*eval*[*e*; *a*] =

[*atom*[*e*] → *cdr*[*assoc*[*e*; *a*]];  
*atom*[*car*[*e*]] → [ *eq*[*car*[*e*]; *quote*] → *cadr*[*e*];  
*eq*[*car*[*e*]; *cond*] → *evcon*[*cdr*[*e*]; *a*];  
*tt* → *apply*[*car*[*e*]; *evlis*[*cdr*[*e*]; *a*]]; *a*]];

*tt* → *apply*[*car*[*e*]; *evlis*[*cdr*[*e*]; *a*]]]

*evcon*[*c*; *a*] =

[*eval*[*caar*[*c*]; *a*] → *eval*[*cadar*[*c*]; *a*];  
*tt* → *evcon*[*cdr*[*c*]; *a*]]

*evlis*[*m*; *a*] =

[*null*[*m*] → *nil*;  
*tt* → *cons*[*eval*[*car*[*m*]; *a*]; *evlis*[*cdr*[*m*]; *a*]]]

■

Die ‘call-by-value’-Implementierung erkennt man im letzten Zweig der Funktion *eval*, der dann verwendet wird, wenn das Argument  $e$  aus einer noch nicht reduzierten Funktion *car*[ $e$ ], angewendet auf die Argumente *cdr*[ $e$ ], besteht. In diesem Fall wird die Auswertungsfunktion *apply* auf die Funktion *car*[ $e$ ] und die durch *evalis* ausgewerteten Argumente angewendet.

### 5.1.4 Inkonsistenzen von LISP

Die Inkorrektheit von Kern-LISP bzgl. der statischen Semantik läßt sich nun durch Angabe einer LISP-Funktion  $fn$  und eines Arguments  $e$ , so daß

$$\mathcal{F}[fn] \mathcal{E}[e] \neq \underline{\text{evalquote}}[\tau fn; (e)]$$

gilt, formal nachweisen.

#### Satz 5.3

*Die LISP-Semantik ist ungleich der statischen Semantik.*

**Beweis:** Sei

$$fn = \lambda[[x]; \text{label } f; \lambda||z]; |z = 0 \longrightarrow x; tt \longrightarrow \lambda[[x]; f[x - 1]][1]]][1]$$

und

$$e = 0 .$$

Sei ferner  $\sigma$  eine Umgebung. Da aus der Syntax bereits ersichtlich ist, ob es sich um einen Ausdruck oder eine Funktion handelt, können hier die Namen  $\mathcal{E}$  bzw.  $\mathcal{F}$  der Semantikfunktionen weggelassen werden, so daß ein Ausdruck der Form

$[\dots]$

für den entsprechenden Ausdruck

$$\mathcal{E}[\dots] \text{ bzw. } \mathcal{F}[\dots]$$

steht.

$$\begin{aligned} 1. \quad & [fn]\sigma \ [e]\sigma \\ &= [\text{label } [\dots][1]]\sigma[0/x] \\ &= [\lambda[[z]; \dots]\sigma[0/x] [\underline{\text{label}} \dots]\sigma[0/x]/f] (1) \text{ Fixpunkteigenschaft} \\ &= [[z = 0 \longrightarrow x; tt \longrightarrow \dots]\sigma[0/x] [\underline{\text{label}} \dots]\sigma[0/x]/f][1/z] \end{aligned}$$

$$\begin{aligned}
 &= [[\lambda[[x]; f[x - 1]][1]]\sigma[0/x] [[\underline{\text{label}} \dots] \sigma[0/x]/f][1/z]] \\
 &= [f[x - 1]]\sigma[0/x] [[\underline{\text{label}} \dots] \sigma[0/x]/f][1/z][1/x] \\
 &= [\underline{\text{label}} \dots] \sigma[0/x](0) \\
 &= 0
 \end{aligned}$$

2. `evalquote[τ fn; (e)]`

$$\begin{aligned}
 &= \underline{\text{apply}}[\tau fn; (0); \underline{\text{nil}}] \\
 &= \underline{\text{apply}}[(\underline{\text{lambda}}(x)((\underline{\text{label}} f(\underline{\text{lambda}}(z)(\underline{\text{cond}}((\underline{\text{eq}} z (\underline{\text{quote}} 0))x) \\
 &\quad (\underline{\text{tt}}((\underline{\text{lambda}}(x) (f(-x(\underline{\text{quote}} 1))))(\underline{\text{quote}} 1)))))(\underline{\text{quote}} 1)); (0); \underline{\text{nil}}] \\
 &= \underline{\text{eval}}[((\underline{\text{label}} f \dots)(\underline{\text{quote}} 1)); ((x.0))] \\
 &= \underline{\text{apply}}[(\underline{\text{label}} f \dots); \underline{\text{evlis}}[((\underline{\text{quote}} 1)); ((x.0))]; ((x.0))] \\
 &= \underline{\text{apply}}[(\underline{\text{label}} f (\underline{\text{lambda}}(z) \dots)); (1); ((x.0))] \\
 &= \underline{\text{apply}}[(\underline{\text{lambda}}(z) \dots); (1); ((f.(\underline{\text{lambda}}(z) \dots))(x.0))] \\
 &= \underline{\text{eval}}[(\underline{\text{cond}} \dots); ((z.1)(f.(\underline{\text{lambda}}(z) \dots))(x.0))] \\
 &= \underline{\text{eval}}[(\underline{\text{lambda}}(z) \dots)(\underline{\text{quote}} 1)); ((z.1)(f.(\underline{\text{lambda}}(z) \dots))(x.0))] \\
 &= \underline{\text{apply}}[(\underline{\text{lambda}}(z) \dots); \underline{\text{evlis}}[((\underline{\text{quote}} 1)); \dots]; ((z.1)(f. \dots)(x.0))] \\
 &= \underline{\text{apply}}[(\underline{\text{lambda}}(z) \dots); (1); ((z.1)(f.(\underline{\text{lambda}}(z) \dots))(x.0))] \\
 &= \underline{\text{eval}}[(f(-x(\underline{\text{quote}} 1))); ((x.1)(z.1)(f.(\underline{\text{lambda}}(z) \dots))(x.0))] \\
 &= \underline{\text{apply}}[f; 0; ((x.1)(z.1)(f.(\underline{\text{lambda}}(z) \dots))(x.0))] \\
 &= \underline{\text{apply}}[\underline{\text{eval}}[f; (\dots)]; 0; ((x.1)(z.1)(f.(\underline{\text{lambda}}(z) \dots))(x.0))] \\
 &= \underline{\text{apply}}[(\underline{\text{lambda}}(z) \dots); 0; ((x.1)(z.1)(f.(\underline{\text{lambda}}(z) \dots))(x.0))] \\
 &= \underline{\text{eval}}[(\underline{\text{cond}} \dots); ((z.0)(x.1)(z.1)(f.(\underline{\text{lambda}}(z) \dots))(x.0))] \\
 &= \underline{\text{eval}}[x:((z.0)(x.1)(z.1)(f.(\underline{\text{lambda}}(z) \dots))(x.0))] \\
 &= 1
 \end{aligned}$$

Q.E.D.

An diesem Beweis erkennt man den Unterschied zwischen statischer und dynamischer Variablenbindung. An das Funktionssymbol  $f$  wird bei statischer Semantik die Funktion  $\llbracket \text{label } f; \dots \rrbracket \sigma$  gebunden, wobei  $\sigma$  die statische Umgebung dieser Funktionsdefinition ist, während der LISP-Interpreter das Tupel  $(f, \dots)$  textmäßig in die a-list schreibt, ohne die aktuelle Umgebung mit festzuhalten. Bei einem Aufruf von  $f$  wird dann diese Funktion bzgl. der dann erstellten a-list, also in der Aufrufumgebung ausgewertet. Nun könnte man meinen, es sei nur eine Frage der Konvention, ob statische oder dynamische Semantik vereinbart wird. Dies ist leider nicht ohne schwerwiegende Konsequenzen der Fall, wie folgende Sätze verdeutlichen.

**Satz 5.4** *Die Semantik von Kern-LISP ist nicht konsistent mit dem Gesetz der Variablenumbenennung.*

**Beweis:** Sei  $fn$  wie im Beweis von Satz 5.3. Durch Umbenennung der inneren Abstraktion nach  $x$  in  $y$  entsteht der Ausdruck

$$fn' = \lambda[[x]; \text{label } f; \lambda[[z]; [z = 0 \rightarrow x; \underline{tt} \rightarrow \lambda[[y]; f[y - 1]][1]]][1]] .$$

Nun läßt sich leicht verifizieren, daß die Gleichung

$$\underline{\text{evalquote}}[\tau fn'; (e)] = 0$$

gilt, während ja im Beweis von Satz 5.3 die Gleichung

$$\underline{\text{evalquote}}[\tau fn; (e)] = 1$$

hergeleitet wurde.

Q.E.D.

**Satz 5.5** *Die Semantik von Kern-LISP ist nicht konsistent mit dem Gesetz der  $\beta$ -Reduktion.*

**Beweis:** Sei wieder  $fn$  wie im Beweis von Satz 5.3. Durch Reduktion des innersten Redexes entsteht der Ausdruck

$$fn'' = \lambda[[x]; \text{label } f; \lambda[[z]; [z = 0 \rightarrow x; \underline{tt} \rightarrow f[0]][1]]] .$$

Nun gilt wieder die Gleichung

$$\underline{\text{evalquote}}[\tau fn''; (e)] = 0 ,$$

während im Beweis von Satz 5.3 die Gleichung

$$\underline{\text{evalquote}}[\tau fn; (e)] = 1$$

gezeigt wurde.

Q.E.D.

Daß es sich bei der Implementierung von LISP mit dynamischer Variablenbindung um ein Versehen handelt, wird deutlich, wenn man im LISP 1.5 Programmer's Manual liest: „The variables in a lambda-expression are dummy or bound variables because systematically changing them does not alter the meaning of the expression“, [87] S. 7. Diese Aussage ist nach Satz 5.4 falsch.

Leider ist dieser Fehler bis heute in allen verbreiteten LISP-Implementierungen, insbesondere auch in den LISP-Maschinen von Symbolics und LMI, vorhanden. Bei der Untersuchung von Programmeigenschaften, muß man daher auf die Definition des Interpreters zurückgreifen, anstatt die Gesetze des  $\lambda$ -Kalküls verwenden zu können. Dies führt insbesondere dazu, daß sich LISP-Programme nicht so leicht verifizieren lassen wie die entsprechenden Ausdrücke des  $\lambda$ -Kalküls. Bereits der kurze Beweis zu Satz 5.3 macht den Unterschied in der Komplexität zwischen  $\lambda$ -Reduktionen und entsprechenden Schritten des Interpreters evalquote deutlich.

## 5.2 FP-Systeme

Backus führt in seinem „ACM Turing Award Lecture“ [9] aus, daß konventionelle Programmiersprachen zu groß, zu komplex und zu unflexibel seien, um heutigen Ansprüchen an Ausdrucksstärke und semantische Klarheit gerecht zu werden. Seiner Meinung nach orientieren sich die Programmiersprachen immer noch zu stark am Konzept des von Neumann-Rechners, der über *einen Kanal in sequentieller Reihenfolge* Informationen zwischen der Zentraleinheit und dem Kernspeicher austauscht. Diese eingeschränkte Möglichkeit der Speichermanipulation nennt er das „von Neumann bottleneck“. Es führt auf der Seite der Sprachen dazu, daß ein Programm eine sequentielle Folge von Zuweisungsbefehlen definiert, anstatt auch von komplexeren Speichermanipulationen Gebrauch zu machen.

Als Alternative zu diesen Konzepten schlägt Backus funktionale Programmierung vor, wobei er das Prinzip verfolgt, aus einigen vorgegebenen Basisprogrammen unter Verwendung von Kombinatoren in hierarchischer Weise neue Programme zu konstruieren. In seiner Sprache, den FP-Systemen besteht ein Programm aus einer Funktionsdefinition ohne Variablen. Alle Funktionen sind vom gleichen Typ, nämlich monadische Abbildungen über Atomen und geschachtelten Listen. Die Datenstruktur ist also die gleiche wie in LISP, nur nennt Backus die Elemente daraus *Objekte* und beschreibt sie in einer abgeänderten Notation.

Ein FP-System besteht aus

- einer Menge  $O$  von *Objekten*,
- einer Menge  $F$  von *Funktionen* über  $O$ . Die Menge  $F$  besteht aus drei Teilmengen: den *primitiven Funktionen*, der Menge  $\mathcal{F}$  von *funktionalen Ausdrücken* und der Menge  $D$  von *Definitionen*.

Im folgenden sollen diese Komponenten der Reihe nach beschrieben werden:

**Objekte** Die Menge  $O$  der Objekte über einer Menge  $A$  von Atomen ist induktiv durch 1. und 2. bestimmt:

1.  $\perp \in O$ ,  $\emptyset \in O$  und  $a \in O$  für alle Atome  $a$  aus  $A$ .
2.  $\langle x_1, \dots, x_n \rangle \in O$  mit  $x_i \in O$ ,  $1 \leq i \leq n$ ,  $n \in \text{IN}$ .

Dabei entspricht das Symbol  $\emptyset$  dem *NIL* aus LISP und bezeichnet die leere Liste. Es ist das einzige Objekt, das sowohl ein Atom als auch eine Liste ist. Die Atome  $T$  und  $F$  bezeichnen die Wahrheitswerte „wahr“ und „falsch“.

**Funktionen** Funktionen sind Abbildungen von  $O$  nach  $O$ . Die Applikation einer Funktion  $f$  auf ein Argument  $x$  wird durch  $f : x$  bezeichnet.

Alle Funktionen aus  $F$  sind stetig und strikt, d.h.  $F$  ist eine Teilmenge von  $[O \rightarrow O]$  mit  $f : \perp = \perp$  für alle  $f$  aus  $F$ .

Zur Definition von Elementen aus  $F$  verwendet Backus eine Variante von McCarthy's bedingter Verzweigung ([83]). So schreibt er

$$p_1 \longrightarrow e_1; \dots; p_n \longrightarrow e_n; e_{n+1}$$

anstelle von McCarthy's Ausdruck

$$(p_1 \longrightarrow e_1, \dots, p_n \longrightarrow e_n, T \longrightarrow e_{n+1}).$$

### Primitive Funktionen

**Selektoren** Jede natürliche Zahl  $s$  dient als Selektor:

$$s : x \equiv x = \langle x_1, \dots, x_n \rangle \& n \geq s \longrightarrow x_s; \perp$$

Jede natürliche Zahl mit dem Zusatz „r“ dient als Rechtsselektor:

$$sr : x \equiv x = \langle x_1, \dots, x_n \rangle \& n \geq s \longrightarrow x_{n+1-r}; \perp$$

**Beispiel:**

$$\underline{2} : \langle A, B, C, D \rangle = B$$

$$\underline{2r} : \langle A, B, C, D \rangle = C$$

$$2 : \langle A \rangle = \perp$$

### Tail-Funktionen

$$tl : x \equiv x = \langle x_1 \rangle \longrightarrow \emptyset; x = \langle x_1, \dots, x_n \rangle \& n > 1 \longrightarrow \langle x_2 \dots x_n \rangle; \perp$$

$$\underline{tlr} : x \equiv x = \langle x_1 \rangle \longrightarrow \emptyset; x = \langle x_1, \dots, x_n \rangle \& n > 1 \longrightarrow \langle x_1 \dots x_{n-1} \rangle; \perp$$

### Identität

$$id : x = x$$

### Tests auf leere Liste, Atom und Gleichheit

null :  $x \equiv x = \emptyset \rightarrow T; x \neq \perp \rightarrow F; \perp$   
atom :  $x \equiv x \in A \rightarrow T; x \neq \perp \rightarrow F; \perp$   
eq :  $x \equiv x = \langle y, z \rangle \& y = z \rightarrow T; x = \langle y, z \rangle \& y \neq z \rightarrow F; \perp$

### Spiegelung

reverse :  $x \equiv x = \emptyset \rightarrow \emptyset; x = \langle x_1, \dots, x_n \rangle \rightarrow \langle x_n, \dots, x_1 \rangle; \perp$

### Verteilungsfunktionen

distl :  $x \equiv x = \langle y, \emptyset \rangle \rightarrow \emptyset;$   
 $x = \langle y, \langle z_1, \dots, z_n \rangle \rangle \rightarrow \langle \langle y, z_1 \rangle, \dots, \langle y, z_n \rangle \rangle; \perp$   
distr :  $x \equiv x = \langle \emptyset, y \rangle \rightarrow \emptyset;$   
 $x = \langle \langle z_1, \dots, z_n \rangle, y \rangle \rightarrow \langle \langle z_1, y \rangle, \dots, \langle z_n, y \rangle \rangle; \perp$

### Längenfunktion

length :  $x = x = \emptyset \rightarrow 0; x = \langle x_1, \dots, x_n \rangle \rightarrow n; \perp$

### Arithmetische Funktionen

$+ : x = x = \langle y, z \rangle \& y, z \in IN \rightarrow y + z; \perp$   
 $- : x = x = \langle y, z \rangle \& y, z \in IN \rightarrow y - z; \perp$   
 $* : x \equiv x = \langle y, z \rangle \& y, z \in IN \rightarrow y * z; \perp$   
 $\div : x \equiv x = \langle y, z \rangle \& y, z \in IN \rightarrow y \div z; \perp$

### Transposition

trans :  $x \equiv x = \langle \emptyset, \dots, \emptyset \rangle \rightarrow \emptyset;$   
 $x = \langle x_1, \dots, x_n \rangle \rightarrow \langle y_1, \dots, y_m \rangle; \perp$

wobei

$x_i = \langle x_{i_1}, \dots, x_{i_m} \rangle$  und  $y_j = \langle x_{1j}, \dots, x_{nj} \rangle$

für alle  $1 \leq i \leq n, 1 \leq j \leq m$ .

### Boolesche Funktionen

and :  $x \equiv x = \langle T, T \rangle \rightarrow T;$   
 $x = \langle T, F \rangle \vee x = \langle F, T \rangle \vee x = \langle F, F \rangle \rightarrow F; \perp$   
or :  $x \equiv x = \langle F, F \rangle \rightarrow F;$   
 $x = \langle T, F \rangle \vee x = \langle F, T \rangle \vee x = \langle T, T \rangle \rightarrow T; \perp$   
not :  $x \equiv x = T \rightarrow F; x = F \rightarrow T; \perp$

### Append-Funktionen

$$\begin{aligned}\underline{\text{apndl}} : x &\equiv x = \langle y, \emptyset \rangle \longrightarrow \langle y \rangle; \\ &x = \langle y, \langle z_1, \dots, z_n \rangle \rangle \longrightarrow \langle y, z_1, \dots, z_n \rangle; \perp \\ \underline{\text{apndr}} : x &\equiv x = \langle \emptyset, y \rangle \longrightarrow \langle y \rangle; \\ &x = \langle \langle z_1, \dots, z_n \rangle, y \rangle \longrightarrow \langle z_1, \dots, z_n, y \rangle; \perp\end{aligned}$$

### Rotationsfunktionen

$$\begin{aligned}\underline{\text{roll}} : x &\equiv x = \emptyset \longrightarrow \emptyset; \\ &x = \langle x_1 \rangle \longrightarrow \langle x_1 \rangle; \\ &x = \langle x_1, \dots, x_n \rangle \& n > 1 \longrightarrow \langle x_2, \dots, x_n, x_1 \rangle; \perp \\ \underline{\text{rollr}} : x &\equiv x = \emptyset \longrightarrow \emptyset; \\ &x = \langle x_1 \rangle \longrightarrow \langle x_1 \rangle; \\ &x = \langle x_1, \dots, x_n \rangle \& n > 1 \longrightarrow \langle x_n, x_1, \dots, x_{n-1} \rangle; \perp\end{aligned}$$

**Funktionale Ausdrücke** Ein funktionaler Ausdruck bezeichnet eine Funktion, deren Parameter nicht nur Objekte sondern auch Funktionen sein können. Backus verwendet die Buchstaben  $f$ ,  $g$  und  $p$  als Metavariable für Funktionen.

### Komposition

$$(f \circ g) : x \equiv f : (g : x)$$

### Konstruktion

$$[f_1, \dots, f_n] . x \equiv \langle f_1 : x, \dots, f_n : x \rangle$$

### Verzweigung

$$(p \longrightarrow f; g) : x \equiv (p : x) = T \longrightarrow f : x; (p : x) = F \longrightarrow g : x; \perp$$

### Konstante Funktionen

$$x : y \equiv y = \perp \longrightarrow \perp; x \text{ für alle } x \in O$$

### Einsetzung

$$\begin{aligned}/f : x &\equiv x = \langle y \rangle \longrightarrow y; \\ &x = \langle x_1, \dots, x_n \rangle \& n > 1 \longrightarrow f : \langle x_1, /f : \langle x_2, \dots, x_n \rangle \rangle; \perp\end{aligned}$$

Wenn  $f$  ein eindeutiges rechtsneutrales Element  $u_f \neq \perp$  besitzt, d.h.  $f : \langle x, u_f \rangle \in \{x, \perp\}$  für alle  $x \in O$ , dann wird diese Definition erweitert:

$$/f : \emptyset = u_f.$$

**Beispiel:**

$$\begin{aligned}
 /+ : < 4, 5, 6 > &= + : < 4, /+ : < 5, 6 > > \\
 &= + : < 4, + : < 5, /+ : < 6 > > > \\
 &= + : < 4, + : < 5, 6 > > \\
 &= 15
 \end{aligned}$$

$$/+ : \emptyset = 0$$

**Allanwendung**

$$\begin{aligned}
 \alpha f : x &\equiv x = \emptyset \longrightarrow \emptyset; \\
 x - < x_1, \dots, x_n > &\longrightarrow < f : x_1, \dots, f : x_n >; \perp
 \end{aligned}$$

**Binär in Monadisch**

$$(\underline{bu} \ f \ x) : y \equiv f : < x, y >$$

**Schleifen**

$$\begin{aligned}
 (\underline{while} \ p \ f) : x &= x = F \longrightarrow x; \\
 x = T &\longrightarrow (\underline{while} \ p \ f) : (f : x); \perp
 \end{aligned}$$

**Definitionen** Eine *Definition* in einem FP-System ist ein Ausdruck der Form

$$\mathbf{Def} \ l \equiv r,$$

wobei die linke Seite  $l$  ein noch nicht verwendetes Funktionssymbol ist und die rechte Seite  $r$  ein funktionaler Ausdruck, in dem  $l$  vorkommen darf.

**Beispiel:**

$$\mathbf{Def} \ \underline{last} \equiv \underline{null} \circ \underline{tl} \longrightarrow 1; \ \underline{last} \circ \underline{tl}$$

Sieht man von den Konventionen in der Notation ab, so gilt für die FP-Systeme: Die Objektsprache ist eine echte Teilmenge der Metasprache. Daher ist die Semantik eines Ausdrucks das durch diesen Ausdruck bezeichnete Objekt, bzw. die durch diesen Ausdruck bezeichnete Funktion.

**Beispiel:** Ein FP-Ausdruck zur Berechnung des Skalarproduktes zweier Vektoren lautet:

$$\text{Def } \underline{sp} \equiv (/+) \circ (\alpha*) \circ \underline{trans}$$

Die Semantik dieses Ausdruckes, angewendet auf das Argument

$$x = << 2, 5, 4 >, < 3, 2, 3 >>,$$

lässt sich nun anhand der Bedeutung der Teilausdrücke wie folgt berechnen:

$$\begin{aligned} \underline{sp} : x &= ((/+) \circ (\alpha*) \circ \underline{trans}) : x \\ &= (/+) : (\alpha* : (\underline{trans} : x)) \\ &= (/+) : (\alpha* : << 2, 3 >, < 5, 2 >, < 4, 3 >>) \\ &= (/+) : < * : < 2, 3 >, * : < 5, 2 >, * : < 4, 3 >> \\ &= (/+) : < 6, 10, 12 > \\ &= 28 \end{aligned}$$

Der entscheidende Vorteil der Verschmelzung zwischen Objekt- und Metasprache liegt darin, daß Aussagen über Programme in der gleichen Sprache geschrieben und bewiesen werden können, in der auch die Programme selbst formuliert sind.

Als Beispiel sei folgendes Lemma bewiesen:

**Lemma:**

$$\underline{apndl} \circ [f \circ g, \alpha f \circ h] = \alpha f \circ \underline{apndl} \circ [g, h]$$

**Beweis:** Sei  $x \in O$ .

**1. Fall**  $h : x$  ist keine Folge, insbesondere nicht die leere Folge  $\emptyset$ . Dann ergeben beide Gleichungsseiten, angewendet auf  $x$ , den Wert  $\perp$ .

**2. Fall**  $h : x = \emptyset$ . Dann gilt:

$$\begin{aligned} \underline{apndl} \circ [f \circ g, \alpha f \circ h] : x &= \underline{apndl} : < f \circ g : x, \emptyset > \\ &= < f : (g : x) > \\ \alpha f \circ \underline{apndl} \circ [g, h] : x &= \alpha f \circ \underline{apndl} : < g : x, \emptyset > \\ &= \alpha f : < g : x > \\ &= < f : (g : x) > \end{aligned}$$

**3. Fall**  $h : x = \langle y_1, \dots, y_n \rangle$ . Dann gilt:

$$\begin{aligned} \underline{\text{apndl}} \circ [f \circ g, \alpha f \circ h] : x &= \underline{\text{apndl}} : \langle f \circ g : x, \alpha f : \langle y_1, \dots, y_n \rangle \rangle \\ &= \langle f : (g : x), f : y_1, \dots, f : y_n \rangle \\ \alpha f \circ \underline{\text{apndl}} \circ [g, h] : x &= \alpha f \circ \underline{\text{apndl}} : \langle g : x, \langle y_1, \dots, y_n \rangle \rangle \\ &= \alpha f : \langle g : x, y_1, \dots, y_n \rangle \\ &= \langle f : (g : x), f : y_1, \dots, f : y_n \rangle \end{aligned}$$

Q.E.D.

Im Artikel von Backus [9] findet sich eine Vielzahl solcher allgemeiner Gesetze und ihre Beweise.

Eine Weiterentwicklung der Sprache der FP-Systeme sind die FFP-Systeme (Formal Systems for Functional Programming), in denen alle Objekte Funktionen darstellen, womit ähnlich wie in LISP universelle Funktionen existieren und neue funktionale Ausdrücke durch ein Programm definiert werden können. Auch hier sei zur ausführlichen Abhandlung auf Backus [9] verwiesen.

Abschließend sei noch erwähnt, daß der Programmierstil von FP- und FFP-Systemen dem Programmierer ein hohes Maß an Vertrautheit mit der Wirkungsweise der Kombinatoren abverlangt. Die entsprechenden Ausdrücke in der  $\lambda$ -Notation sind zwar etwas länger, lassen sich aber leichter lesen, weil die Objekte benannt werden können und der Ausdruck unmittelbar beschreibt, wie auf ihnen operiert wird.

Zur Illustration sei der Ausdruck für das Skalarprodukt aufgeführt:

$$\begin{aligned} \underline{\text{sp}} <><> &= 0 \\ \underline{\text{sp}} x.\bar{x} y.y &= x * y + (\underline{\text{sp}} x y) \end{aligned}$$

Eine Sprache, die auf solchen Gleichungssystemen aufbaut, ist die von Turner entwickelte Sprache Miranda, die im nächsten Abschnitt behandelt wird.

### 5.3 Programmieren mit rekursiven Gleichungssystemen

In den vergangenen zehn Jahren wurde eine Reihe funktionaler (applikativer) Programmiersprachen entwickelt, die sich durch eine klare Semantik auszeichnen, d.h. sie verwenden aus der Mathematik bekannte Notationen, die dann auch in der vertrauten Weise interpretiert werden. Damit ist die Verschmelzung von Syntax und

Semantik erreicht. Im wesentlichen besteht ein funktionales Programm aus einem rekursiven Gleichungssystem mit Funktionssymbolen und formalen Parametern auf den linken Seiten und applikativen Ausdrücken bzw.  $\lambda$ -Ausdrücken auf den rechten Seiten. Die beiden Seiten einer solchen Gleichung aus einem funktionalen Programm bezeichnen also stets auch semantisch gleiche Objekte. Daher kann etwa in einem Beweis ein Ausdruck, der der linken Seite entspricht, durch den entsprechenden Ausdruck der rechten Seite ersetzt werden und umgekehrt. Ein weiterer Vorteil funktionaler Sprachen im Vergleich zu den „von Neumann-Sprachen“ ist wie bei den FP-Systemen ihre Nähe zur Spezifikation, da im Programm nicht mehr ein sequentieller Kontrollfluß definiert werden muß. Der Nachteil der funktionalen Programmierung ist die aufwendige Implementierung auf herkömmlichen Rechnern, die zu erheblicher Ineffizienz führt. Unter Verwendung neuer Technologien, insbesondere die Ausnutzung von VLSI und Parallelität, werden derzeit alternative Rechnermodelle entwickelt, die diesen Nachteil aufheben sollen.

Als Prototyp der Klasse der funktionalen Programmiersprachen soll in diesem Abschnitt die von Turner entwickelte Sprache Miranda vorgestellt werden (s. [122], [124], [125] und [126]). Weitere Sprachen dieser Art sind u.a. die Sprache HOPE von Burstall [22] und der funktionale Kern der Sprache ML von Milner et al. [93].

Die Sprache Miranda unterscheidet sich von Kern-LISP in vierfacher Hinsicht:

1. Die Syntax ist durch die generelle Verwendung von Gleichungen und anderen Konventionen erheblich benutzerfreundlicher.
2. Die Sprache enthält einige mengentheoretische und prädikatenlogische Konzepte.
3. Die Sprache ist streng getypt und erlaubt die Verwendung abstrakter Datentypen.
4. Die Semantik ist genau die statische Semantik rekursiver Gleichungssysteme.

Während die drei ersten Unterschiede unerhebliche Modifikationen sind, die sich auch in die Sprache LISP integrieren ließen, beinhaltet der vierte Punkt den entscheidenden Unterschied, der Miranda zu einer alternativen Sprache macht.

Da an dieser Stelle bereits die formalen Definitionen zweier funktionaler Sprachen mit der Eigenschaft, Objektsprache gleich Metasprache, vorliegen (getypter  $\lambda$ -Kalkül und Kern-LISP mit statischer Semantik), soll die Sprache Miranda an Beispielen informell vorgestellt werden, um die Brauchbarkeit und Eleganz solcher alternativen Sprachen zu illustrieren.

Die *Datenobjekte* von Miranda umfassen die primitiven Bereiche der Zahlen, der Wahrheitswerte und der Worte sowie die zusammengesetzten Bereiche der Listen und Verbunde (records). Variablen sind Worte über den kleinen lateinischen Buchstaben.

**Beispiele:**

4, 7.36, -28	sind Zahlen
<i>True</i> , <i>False</i>	sind Wahrheitswerte
"Erich", "Das ist ein Satz"	sind Worte
<i>x</i> , <i>y</i> , <i>a</i> , <i>wer</i>	sind Variablen
[], [a, b, c], [[0], [1]]	sind Listen
(a, "Jones", 3, <i>True</i> )	ist ein Verbund

Folgende *Operationen* sind auf *Listen* erklärt:

- Der Infixoperator ! zur Indizierung. Wenn  $L$  eine Liste ist und  $i$  eine natürliche Zahl, dann bezeichnet der Ausdruck  $L!i$  das  $(i-1)$ -te Element der Liste  $L$ , z.B.

$$[l_0, l_1, l_2, l_3]!2 = l_2$$

- Der Präfixoperator # zur Bestimmung der Länge einer Liste, z.B.

$$\# [l_0, l_1, l_2, l_3] = 4$$

- Der Infixoperator : zur Realisierung der cons-Funktion, z.B.

$$0 : [1, 2, 3] = [0, 1, 2, 3]$$

- Der Infixoperator ++ zur Konkatenation von Listen, z.B.

$$[0, 1, 2] + + [3, 4] = [0, 1, 2, 3, 4]$$

- Der Infixoperator -- zur Differenz von Listen, z.B.

$$[0, 1, 2, 3, 4] -- [1, 3] = [0, 2, 4]$$

Ferner gibt es in Miranda eine Kurzschreibweise .. für Listen, deren Elemente eine arithmetische Folge bilden, z.B.

$$[0..4] = [0, 1, 2, 3, 4]$$

$$[1, 3..9] = [1, 3, 5, 7, 9]$$

Da in Miranda auch unendliche Listen als Objekte zugelassen sind, kann die rechte Listenbegrenzung auch weggelassen werden. So bezeichnet der Ausdruck

$$[3..]$$

die Liste aller natürlichen Zahlen größer gleich 3 und der Ausdruck

$$[1, 3..]$$

die Liste aller ungeraden Zahlen.

*Funktionen* werden in Miranda durch Gleichungen derart definiert, daß der Funktionsname, gefolgt von den formalen Parametern, auf der linken Seite steht und der definierende Ausdruck auf der rechten. Folgende Konventionen verbessern die Übersichtlichkeit:

- Funktionale Applikation wird durch Hintereinanderschreibung notiert und linksassoziativ vereinbart.
- Formale Parameter können strukturiert sein (pattern), z.B.

$$x : l .$$

- Anstelle von if - then - else Ausdrücken sind zu einer linken Gleichungsseite mehrere rechte Seiten erlaubt, die durch disjunkte Prädikate (guards) geschützt sind.

### Beispiele:

Die Gleichung

$$fac\ n = \underline{product}\ [1..n]$$

definiert die Fakultätsfunktion, wobei product eine Systemfunktion ist, die alle Elemente einer Liste multipliziert.

Ein Gleichungssystem zur Definition der Fakultätsfunktion ohne Verwendung der Systemfunktion product lautet:

$$\begin{aligned} fac\ 0 &= 1 \\ fac\ (n + 1) &= (n + 1) * fac\ n \end{aligned}$$

Die Funktion zur Berechnung des größten gemeinsamen Teilers (gcd) kann durch folgende Gleichungen definiert werden:

$$\begin{aligned} gcd\ a\ b &- gcd\ (a - b)\ b, \ a > b \\ &= gcd\ a\ (b - a), \ a < b \\ &= a, \ a = b \end{aligned}$$

Ein Gleichungssystem zum Sortieren durch direktes Einfügen lautet:

$$\begin{aligned}
 \text{sort} [] &= [] \\
 \text{sort} (a : x) &= \text{insert } a (\text{sort } x) \\
 \text{insert } a [] &= [a] \\
 \text{insert } a (b : x) &= a : b : x, a \leq b \\
 &= b : \text{insert } a x, a > b
 \end{aligned}$$

Folgendes Gleichungssystem demonstriert die Verwendung von Funktionalen:

$$\begin{aligned}
 \text{answer} [] &= \text{twice twice twice suc } 0 \\
 \text{twice } f x &= f (f x) \\
 \text{suc } x &= x + 1
 \end{aligned}$$

Der bis hierher vorgestellte Sprachumfang entspricht bis auf triviale syntaktische Variationen der Metasprache der denotationellen Semantikmethode und ist bereits gut als Programmiersprache geeignet.

In der Sprache Miranda gibt es über den bisher vorgestellten Rahmen hinaus die Möglichkeit, Mengen über Prädikate zu definieren. Dadurch kann vielfach eine explizite Rekursion vermieden werden, was sehr zur Durchschaubarkeit der Programme beiträgt. Basierend auf der Mengentheorie von Zermelo und Fraenkel (s. [43]) sind in Miranda ZF-Ausdrücke zur Definition von Mengen erlaubt. Ein ZF-Ausdruck hat stets die Form

$$[e \mid p],$$

wobei  $e$  ein Ausdruck ist, und  $p$  eine nichtleere Folge von Generatoren und booleschen Ausdrücken über den in  $e$  frei vorkommenden Variablen. Dabei wird verlangt, daß jede solche Variable durch einen beschränkten Allquantor (Generator) gebunden ist.

**Beispiele:** Der ZF-Ausdruck

$$[n * n \mid n \leftarrow [1..100]]$$

hat die Liste der ersten hundert Quadratzahlen (in aufsteigender Reihenfolge) als Wert.

**Der Ausdruck**

$$[[i, j] \mid i \leftarrow [1..9]; j \leftarrow [1..9]; i + j = 10]$$

definiert alle Paare natürlicher Zahlen, deren Summe gleich 10 ist. Wünscht man nur die geordneten Paare zu definieren, so schreibt man

$$[[i, j] \mid i \leftarrow [1..9]; j \leftarrow [i..9]; i + j = 10].$$

Schließlich sei noch ein Gleichungssystem zur Erzeugung aller Permutationen einer gegebenen Liste aufgeführt:

$$\begin{aligned} \text{perm } [] &= [[]] \\ \text{perm } x &= |a : y \mid a \leftarrow x; y \leftarrow \text{perms } (x - [a])| . \end{aligned}$$

Abschließend soll zur Illustration der Nähe von funktionalen Programmen zur Spezifikation das 8-Damen-Problem in Miranda gelöst werden. Dabei sollen acht Damen so auf ein Schachbrett gestellt werden, daß keine davon eine andere schlagen kann.

Die Idee des Algorithmus' lautet: Setze von links nach rechts je eine Dame auf die nächste Spalte und prüfe, ob sie in bezug auf alle vorher gesetzten sicher ist. Eine geeignete Darstellung des Schachbrettes, besteht aus einer Liste von natürlichen Zahlen

$$[n_0, \dots, n_7],$$

wobei in der  $i$ -ten Spalte eine Dame auf der  $n_i$ -ten Zeile steht.

Während man in einer herkömmlichen Programmiersprache einen „back-track“-Algorithmus formulieren würde, der bei Nichtauftreffen einer sicheren Position die vorher gesetzte Dame neu aufstellen würde, lässt sich dieses Vorgehen in Miranda implizit ausdrücken, und somit entsteht ein übersichtliches, sich selbst erklärendes Programm:

$$\begin{aligned} \text{queens } 0 &= [[]] \\ \text{queens } (n + 1) &= [q : b \mid q \leftarrow [0..7]; b \leftarrow \text{queens } n; \text{safe } q \ b] \\ \text{safe } q \ b &= \text{and } [\sim \text{checks } q \ b \ i \mid i \leftarrow [0..\#b - 1]] \\ \text{checks } q \ b \ i &= q - b!i \vee \text{abs}(q - b!i) = i \end{aligned}$$

Eine Berechnung des Ausdrucks

$$\text{queens } 8$$

ergibt dann die Liste aller Lösungen des 8-Damen-Problems.

Der Implementierung von Miranda unterliegt das Prinzip der „verzögerten Auswertung“ (lazy evaluation) von Henderson und Morris [45], wodurch sicher gestellt wird, daß kein Teilausdruck ausgewertet wird, bevor nicht feststeht, daß sein Wert wirklich zur Berechnung des Endergebnisses gebraucht wird. Informell lässt sich diese Strategie wie folgt beschreiben: Man wertet einen Ausdruck stets von links nach rechts aus. Trifft man auf ein Funktionssymbol, so wird die entsprechende

rechte Gleichungsseite ersetzt, wobei die formalen Parameter durch Zeiger auf die aktuellen Werte ersetzt werden. Trifft man auf einen solchen Zeiger, reduziert man den dazugehörigen Ausdruck so weit wie möglich und setzt dann erst eine Kopie an die Stelle des Zeigers.

Dieses Vorgehen führt dazu, daß keine unnötigen Reduktionen gemacht werden und sogar unendliche Listen behandelt werden können.

**Beispiele:**

```

ones  =  1 : ones
repeat a  =  x
           where x = a : x
nats  ~  [0..]
odds  =  [1,3..]
squares  =  [n * n | n ← [0..]]
primes  =  sieve [2..]
           where sieve (p : x) = p : sieve [n | n ← x; n mod p > 0]
```

Weitere Konzepte von Miranda, auf die hier nicht weiter eingegangen werden soll, sind ein polymorphes Typkonzept im Sinne von Damas und Milner [30], benutzerdefinierte Typen und abstrakte Datentypen.

## **Kapitel 6**

### **Anwendungen der denotationellen Semantik bei der Implementierung höherer Programmiersprachen**

Die konventionelle Methode der Implementierung höherer Programmiersprachen ist die Erstellung von Compilern zur Übersetzung von Benutzerprogrammen in entsprechende Maschinenprogramme. In gewissem Sinne legt ein Compiler bereits die Semantik einer Sprache fest. Man kann nämlich vereinbaren, daß die Bedeutung eines Programms gerade diejenige Zustandstransformation ist, die durch das entsprechende Maschinenprogramm definiert wird. Natürlich ist ein solches Vorgehen i.a. unbefriedigend, da die gleiche Sprache bzgl. verschiedener Rechner und verschiedener Compiler immer dieselbe Bedeutung haben sollte, ein Äquivalenzbeweis wegen der Komplexität der Übersetzerprogramme und wegen fehlender Formalisierung der Semantik der Maschinensprachen aber meist nicht möglich ist. Liegt hingegen eine abstrakte formale Semantikbeschreibung einer Sprache vor, gibt es unterschiedliche Ansätze, daraus systematisch zu einer Implementierung zu gelangen.

Im nächsten Abschnitt werden solche Methoden besprochen, die durch Anwendung geeigneter Transformationen auf die Formeln der denotationellen Semantik eine automatische Codeerzeugung für konventionelle Maschinen ermöglichen. In Abschnitt 6.2 wird eine Implementierungsart erläutert, bei der zunächst die  $\lambda$ -Ausdrücke der Semantikdefinitionen in Ausdrücke der kombinatorischen Logik übersetzt werden, die dann auf einem Rechner mit spezieller Kellerarchitektur interpretiert werden können. Schließlich sollen in Abschnitt 6.3 drei Ansätze zur direkten Implementierung funktionaler Sprachen auf innovativen Rechnerarchitekturen vorgestellt werden.

Eine ausführliche und formale Abhandlung des Themas dieses Kapitels würde den Rahmen einer Semantikvorlesung und damit dieses Buches sprengen. Daher soll hier nur informell ein Überblick gegeben werden. Der interessierte Leser sei auf die Literatur verwiesen.

## 6.1 Systematische Codeerzeugung aus der Standardsemantik

Betrachtet man die Semantikfunktionen etwa der Sprache  $PASCAL_0$ , so könnte man zunächst vermuten, daß sich auf sehr einfache Weise daraus Code erzeugen läßt.

Der  $\lambda$ -Ausdruck zur Definition der Semantik der Zuweisung lautet:

$$\mathcal{C}[E_1 := E_2]\rho c = \mathcal{E}[E_1]\rho; LOC? \lambda l. \mathcal{R}[E_2]\rho; \underline{update} \ l \ c.$$

Hat man nun bereits Prozeduren zur Erzeugung von Code für die Berechnung der  $\mathcal{E}$ - bzw.  $\mathcal{R}$ -Semantik von Ausdrücken und Ablage der Ergebnisse in den Registern  $l$  bzw.  $l'$ , so kann man diese wie folgt verbinden:

- Generiere den Code zur Berechnung von  $E_1$  und zum Laden des Ergebnisses nach  $l$ .
- Schreibe Code zum Test, ob in  $l$  eine Adresse steht.

Für den positiven Fall:

- Erzeuge den Code zur Berechnung von  $E_2$  und zum Laden des Ergebnisses nach  $l'$ .
- Schreibe Code zum Laden des Inhaltes von  $l'$  in den Speicherplatz, dessen Adresse in  $l$  steht.

Für den negativen Fall:

- Schreibe Code für eine entsprechende Fehlermeldung.

Die Semantik der Hintereinanderausführung von Anweisungen

$$\mathcal{C}[C_1; C_2]\rho c = \mathcal{C}[C_1]\rho; \mathcal{C}[C_2]\rho c$$

legt ebenfalls die Idee nahe, einfach induktiv den Code zur Ausführung von  $C_1$  vor den Code zur Ausführung von  $C_2$  zu schreiben.

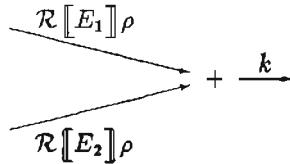
Leider ist dieses Vorgehen nicht unmittelbar möglich, da die logische Struktur der  $\lambda$ -Ausdrücke zur Definition der Semantikfunktion nicht immer linear ist, sondern

Bäumen entspricht.

Die Ausdrucksemantik für die Addition sieht folgendermaßen aus:

$$\mathcal{E}[[E_1 + E_2]]\rho k = \mathcal{R}[[E_1]]\rho \lambda e_1. \mathcal{R}[[E_2]]\rho \lambda e_2. k(e_1 + e_2).$$

Die logische Struktur der rechten Seite dieser Gleichung lässt sich wie folgt verdeutlichen:



Würde man den Code zur Berechnung von  $E_1$  ohne weitere Vorkehrungen vor den Code zur Berechnung von  $E_2$  schreiben und dann entsprechende Befehle zur Ausführung der Addition und der Fortsetzung  $k$ , so würde bei der Berechnung von  $E_2$  das Ergebnis der Berechnung von  $E_1$  fälschlicherweise überschrieben.

Wand hat in [130] eine Methode beschrieben, wie man unter Verwendung spezieller Kombinatoren die logische Struktur der  $\lambda$ -Ausdrücke, die in den Semantikfunktionen verwendet werden, linearisieren kann. Dies sei hier zur Illustration für die Additionsausdrücke durchgeführt.

Die Syntax für Additionsausdrücke lautet:

$$E ::= I \mid E_1 + E_2 .$$

Unter der Annahme, daß nur solche Identifier  $I$  verwendet werden, die als Speicheradressen vereinbart sind, vereinfacht sich die Semantikfunktion  $E$  wie folgt:

$$\begin{aligned}\mathcal{E} : EXP &\longrightarrow EC \longrightarrow CC \\ \mathcal{E}[I]ks &= k(sI) \\ \mathcal{E}[E_1 + E_2]k &= \mathcal{E}[E_1] \lambda e_1. \mathcal{E}[E_2] \lambda e_2. k(e_1 + e_2) .\end{aligned}$$

Will man Additionsausdrücke berechnen, ohne sie in den Kontext eines Programms einzubetten, muß man die Semantikfunktion  $\mathcal{P}$  auch für Ausdrücke erklären:

$$\mathcal{P}[E] = \mathcal{E}[E] \lambda es.e .$$

Zur kombinatorischen Schreibweise dieser drei Semantikgleichungen definiert man drei Hilfsfunktionen:

$$\begin{aligned}\underline{\text{halt}} &= \lambda es.e \\ \underline{\text{fetch}} &= \lambda Iks. k(sI)s \\ \underline{\text{add1}} &= \lambda abk. a\lambda e_1. b\lambda e_2. k(e_1 + e_2)\end{aligned}$$

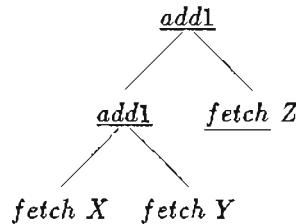
Jetzt wird die Baumstruktur der Semantik von Additionsausdrücken noch deutlicher:

$$\begin{aligned}\mathcal{P}[E] &= \mathcal{E}[E] \text{ } \underline{\text{halt}} \\ \mathcal{E}[I] &= \underline{\text{fetch }} I \\ \mathcal{E}[E_1 + E_2] &= \underline{\text{add1 }} \mathcal{E}[E_1] \mathcal{E}[E_2].\end{aligned}$$

Als Beispiel sei eine Reduktionsfolge angegeben, welche die Semantik des Ausdrucks  $(X + Y) + Z$  in einen kombinatorischen Ausdruck überführt:

$$\begin{aligned}\mathcal{E}[(X + Y) + Z] &= \underline{\text{add1 }} \mathcal{E}[X + Y] \mathcal{E}[Z] \\ &= \underline{\text{add1 }} (\underline{\text{add1 }} \mathcal{E}[X] \mathcal{E}[Y]) \underline{\text{fetch }} Z \\ &= \underline{\text{add1 }} (\underline{\text{add1 }} \underline{\text{fetch }} X \underline{\text{fetch }} Y) \underline{\text{fetch }} Z\end{aligned}$$

Dieser Kombinatorausdruck entspricht folgendem Baum:



Solche Ausdrücke lassen sich zwar leicht über eine Kellerimplementierung interpretieren, sind aber zur Codeerzeugung noch nicht unmittelbar geeignet.

Der nächste Schritt ergibt sich aus der Beobachtung, daß in der ursprünglichen Gleichung für  $\mathcal{E}[E_1 + E_2]$  die  $\lambda$ -Variablen  $e_1$  und  $e_2$  in den Operandenteil einer Applikation eingehen. Dies legt die Verwendung des Kombinators  $B$  nahe:

$$B = \lambda abx.a(bx).$$

Verallgemeinert man den Kombinator  $B$  für  $k \geq 0$  zu

$$B_k = \lambda(a, b)x_1 \cdots x_k.a(bx_1 \cdots x_k),$$

so läßt sich die Semantikfunktion  $\mathcal{E}$  für Additionen wie folgt formulieren:

$$\begin{aligned}\mathcal{E}[E_1 + E_2]k &= \mathcal{E}[E_1] \lambda e_1. \mathcal{E}[E_2] \lambda e_2. k(e_1 + e_2). \\ &= B_1(\mathcal{E}[E_1], \lambda ke_1. \mathcal{E}[E_2] \lambda e_2. k(e_1 + e_2)) \\ &= B_1(\mathcal{E}[E_1], B_2(\mathcal{E}[E_2], \lambda ke_1 e_2. k(e_1 + e_2))).\end{aligned}$$

Mit der Hilfsfunktion

$$\underline{\text{add}} = \lambda ke_1 e_2. k(e_1 + e_2)$$

lassen sich die folgenden Gleichungen ableiten:

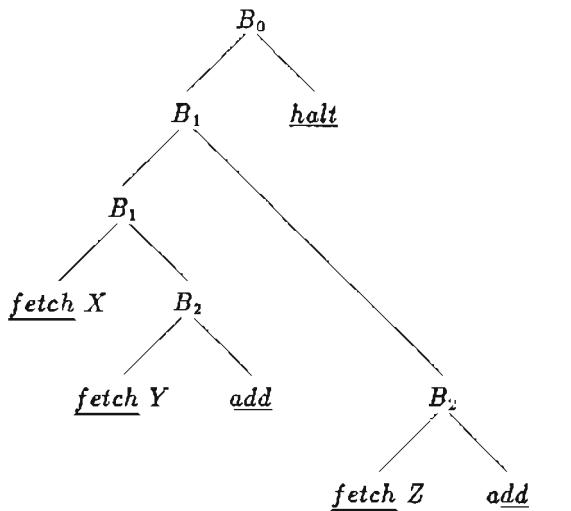
$$\begin{aligned}\mathcal{P}[\![E]\!] &= B_0(\mathcal{E}[\![E]\!], \underline{\text{halt}}) \\ \mathcal{E}[\![I]\!] &= \underline{\text{fetch }} I \\ \mathcal{E}[\![E_1 + E_2]\!] &= B_1(\mathcal{E}[\![E_1]\!], B_2(\mathcal{E}[\![E_2]\!], \underline{\text{add}})) .\end{aligned}$$

Diese Formulierung entspricht einer Baumdarstellung, wobei interne Knoten mit  $B_i$ 's markiert sind und die Blätter mit halt, fetch  $I$  und add.

Beispiel:

$$\begin{aligned}\mathcal{P}[\![(X + Y) + Z]\!] &= B_0(\mathcal{E}[\![(X + Y) + Z]\!], \underline{\text{halt}}) \\ &= B_0(B_1(\mathcal{E}[\![X + Y]\!], B_2(\mathcal{E}[\![Z]\!], \underline{\text{add}})), \underline{\text{halt}}) \\ &= B_0(B_1(\mathcal{E}[\![X + Y]\!], B_2(\underline{\text{fetch }} Z, \underline{\text{add}})), \underline{\text{halt}}) \\ &= B_0(B_1(B_1(\mathcal{E}[\![X]\!], B_2(\mathcal{E}[\![Y]\!], \underline{\text{add}})), B_2(\underline{\text{fetch }} Z, \underline{\text{add}})), \underline{\text{halt}}) \\ &= B_0(B_1(B_1(\underline{\text{fetch }} X, B_2(\underline{\text{fetch }} Y, \underline{\text{add}})), B_2(\underline{\text{fetch }} Z, \underline{\text{add}})), \underline{\text{halt}}) ,\end{aligned}$$

als Baum dargestellt:



Diese Umformung allein ist im Hinblick auf eine Codeerzeugung noch nicht ausreichend. Die Kombinatoren  $B_i$  sind jedoch rechtsassoziativ, und diese Eigenschaft lässt sich zur Linearisierung der Struktur der Ausdrücke ausnutzen.

**Lemma 6.1** Sei  $k \geq 0$  und  $p \geq 1$ .

Es gilt:

$$B_k(B_p(a, b), c) = B_{k+p-1}(a, B_k(b, c)) .$$

**Beweis:**

$$\begin{aligned} & B_k(B_p(a, b), c)x_1 \cdots x_{k+p-1} \\ &= B_p(a, b)(cx_1 \cdots x_k)x_{k+1} \cdots x_{k+p-1} \\ &= a(b(cx_1 \cdots x_k)x_{k+1} \cdots x_{k+p-1}) \text{ da } p \geq 1 \\ &= a(B_k(b, c)x_1 \cdots x_kx_{k+1} \cdots x_{k+p-1}) \\ &= B_{k+p-1}(a, B_k(b, c))x_1 \cdots x_{k+p-1} \end{aligned}$$

Q.E.D.

Wand zeigt in [130] sogar die volle Assoziativität. An dieser Stelle genügt jedoch die Rechtsassoziativität für die gewünschte Transformation der Additionsbäume in lineare Form. Diese Transformation heißt **rot** und ist wegen Lemma 6.1 semantikhaltend.

$$\begin{aligned} \text{rot } [B_k(B_p(a, b), c)] &= \text{rot } [B_{k+p-1}(a, B_k(b, c))] \\ \text{rot } [B_i(\underline{\text{fetch}} I, a)] &= B_i(\underline{\text{fetch}} I, \text{rot } a) \\ \text{rot } [B_i(\underline{\text{add}}, a)] &= B_i(\underline{\text{add}}, \text{rot } a) \\ \text{rot } \underline{\text{halt}} &= \underline{\text{halt}} \\ \text{rot } \underline{\text{add}} &= \underline{\text{add}} \end{aligned}$$

Die Semantikgleichungen lassen sich nun für die Additionsausdrücke folgendermaßen formulieren:

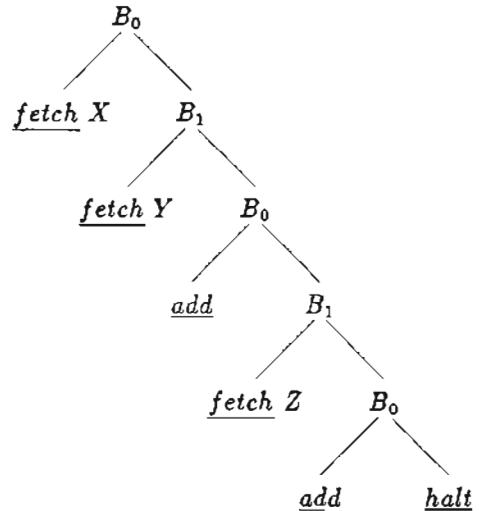
$$\begin{aligned} \mathcal{P}[E] &= \text{rot } [B_0(\mathcal{E}[E], \underline{\text{halt}})] \\ \mathcal{E}[I] &= \underline{\text{fetch}} I \\ \mathcal{E}[E_1 + E_2] &= B_1(\mathcal{E}[E_1], B_2(\mathcal{E}[E_2], \underline{\text{add}})) . \end{aligned}$$

**Beispiel:**

$$\begin{aligned} \mathcal{P}[(X + Y) + Z] &= \\ &= \text{rot } [B_0(\mathcal{E}[(X + Y) + Z], \underline{\text{halt}})] \\ &= \text{rot } [B_0(B_1(\mathcal{E}[(X + Y)], B_2(\mathcal{E}[Z], \underline{\text{add}})), \underline{\text{halt}})] \\ &= \text{rot } [B_0(B_1(B_1(\mathcal{E}[X]), B_2(\mathcal{E}[Y], \underline{\text{add}})), B_2(\underline{\text{fetch}} Z, \underline{\text{add}})), \underline{\text{halt}})] \end{aligned}$$

$$\begin{aligned}
&= \text{rot} [B_0 (B_1 (\underbrace{B_1 (\underline{\text{fetch}} X, B_2 (\underline{\text{fetch}} Y, \underline{\text{add}})), \underbrace{B_2 (\underline{\text{fetch}} Z, \underline{\text{add}})}_{b}), \underline{\text{halt}})] \\
&= \text{rot} [B_0 (B_1 (\underbrace{\underline{\text{fetch}} X}_{a}, \underbrace{B_2 (\underline{\text{fetch}} Y, \underline{\text{add}})}_{b}), \underbrace{B_0 (B_2 (\underline{\text{fetch}} Z, \underline{\text{add}}), \underline{\text{halt}})}_{c})] \\
&= \text{rot} [B_0 (\underline{\text{fetch}} X, B_0 (\underbrace{B_2 (\underline{\text{fetch}} Y, \underline{\text{add}})}_{a}, B_0 (B_2 (\underline{\text{fetch}} Z, \underline{\text{add}}), \underline{\text{halt}})))] \\
&= B_0 (\underline{\text{fetch}} X, \text{rot} [B_0 (B_2 (\underbrace{\underline{\text{fetch}} Y}_{a}, \underbrace{\underline{\text{add}}}_{b}), B_0 (B_2 (\underline{\text{fetch}} Z, \underline{\text{add}}), \underline{\text{halt}}))]) \\
&= B_0 (\underline{\text{fetch}} X, \text{rot} [B_1 (\underline{\text{fetch}} Y, B_0 (\underline{\text{add}}, B_0 (B_2 (\underline{\text{fetch}} Z, \underline{\text{add}}), \underline{\text{halt}})))]) \\
&= B_0 (\underline{\text{fetch}} X, B_1 (\underline{\text{fetch}} Y, \text{rot} [B_0 (\underline{\text{add}}, B_0 (\underbrace{B_2 (\underline{\text{fetch}} Z, \underline{\text{add}})}_{a}, \underline{\text{halt}}))])) \\
&= B_0 (\underline{\text{fetch}} X, B_1 (\underline{\text{fetch}} Y, B_0 (\underline{\text{add}}, \text{rot} [B_0 (B_2 (\underline{\text{fetch}} Z, \underline{\text{add}}), \underline{\text{halt}})]))) \\
&= B_0 (\underline{\text{fetch}} X, B_1 (\underline{\text{fetch}} Y, B_0 (\underline{\text{add}}, \text{rot} [B_1 (\underline{\text{fetch}} Z, B_0 (\underline{\text{add}}, \underline{\text{halt}}))]))) \\
&= B_0 (\underline{\text{fetch}} X, B_1 (\underline{\text{fetch}} Y, B_0 (\underline{\text{add}}, B_1 (\underline{\text{fetch}} Z, \text{rot} [B_0 (\underline{\text{add}}, \underline{\text{halt}})])))) \\
&= B_0 (\underline{\text{fetch}} X, B_1 (\underline{\text{fetch}} Y, B_0 (\underline{\text{add}}, B_1 (\underline{\text{fetch}} Z, B_0 (\underline{\text{add}}, \underline{\text{halt}}))))) ,
\end{aligned}$$

als Baum dargestellt:



Aus diesem Ausdruck kann nun auf einfache Weise Code abgeleitet werden, indem der den Blättern entsprechende Code erzeugt und hintereinandergehängt wird.

Wand vervollständigt in [130] diese Methode für eine Beispielsprache mit Prozeduren.

Es gibt eine Vielzahl weiterer Ansätze der systematischen Codeerzeugung aus der denotationellen Semantik. Ohne Anspruch auf Vollständigkeit seien hier einige davon genannt:

Raskovsky und Collier entwickeln in [104] eine Zwischenstufe, die IDS (Implementation Denotational Semantics), auf der spezielle Strategien für die Implementierung formuliert werden können. Ihr compilererzeugendes System arbeitet dementsprechend in zwei Phasen:

1. Übersetzung von der abstrakten Standardsemantik in die Implementierungssemantik
2. Codeerzeugung aus der Implementierungssemantik

Während die erste Phase im wesentlichen von Hand gemacht werden muß, wird die zweite in starkem Maße vom System unterstützt. Zu jeder IDS-Gleichung wird eine BCPL-Prozedur erzeugt, die dann zur Codegenerierung verwendet werden kann. Die Äquivalenz zwischen der Standardsemantik und der IDS läßt sich mit bekannten Beweismethoden nachweisen. Eine Äquivalenz zwischen der IDS und den entsprechenden BCPL-Prozeduren wird nicht gezeigt. Man kann jedoch davon ausgehen, daß die von diesem System erzeugten Compiler wegen der systematischen Erzeugung recht zuverlässig sind.

Ein ähnlicher Ansatz wird auch von Bjørner [15], Jones [57] sowie Milne und Strachey [90] verfolgt. Hier werden insbesondere in der ersten Phase sehr allgemeine Softwareentwicklungsprinzipien eingesetzt, wie z.B. die schrittweise Verfeinerung abstrakter Datentypen (s. Hoare [49]).

Eine Methode, auch die zweite Phase formal korrekt zu entwickeln, wird von Ganzinger in [36] und [37] vorgestellt. Sein System arbeitet in drei Phasen:

1. Modifikation der denotationellen Semantik in eine Verallgemeinerung attributierter Grammatiken.
2. Eine Analyse der Abhängigkeiten zwischen Argumenten und Ergebnissen der Semantikfunktionen liefert die Unterscheidung von statischer und dynamischer Semantik. Es entsteht eine attributierte Grammatik zur Formalisierung der statischen Semantik und eine Beschreibung der Interpretation eines attribuierten Programmabbaumes unter gegebenen Eingabedaten.
3. Transformation in ein Programm zur Codegenerierung.

Ein weiterer Ansatz, der methodisch sehr elegant ist, in der technischen Durchführung aber sehr aufwendig, ist die algebraische Methode, die auf Morris [94] zurückzuführen ist. Der Grundgedanke dabei läßt sich wie folgt zusammenfassen: Ausgehend von einer höheren Programmiersprache *L* und einer Maschinensprache

$T$  mit Semantikfunktionen in die Bereiche  $M$  bzw.  $U$ , definiert man zu einem gegebenen Compiler  $\gamma$  eine Codierungsfunktion  $\delta$  (oder eine Decodierungsfunktion  $\bar{\delta}$ ) und zeigt, daß folgendes Diagramm kommutiert:

$$\begin{array}{ccc} L & \xrightarrow{\gamma} & T \\ \downarrow [] & & \downarrow [] \\ M & \xrightarrow{\delta} & U \\ & \xrightarrow{\bar{\delta}} & \end{array}$$

Von der Gruppe ADJ wird in [120] die Programmiersprache  $L$  als  $G$ -Algebra aufgefaßt, wobei  $G$  die abstrakte Syntax von  $L$  ist (jeder Syntaxregel entspricht eine algebraische Operation). Sowohl die Semantikfunktionen als auch der Compiler werden als Homomorphismen definiert. Da die Algebra  $L$  initial in der Klasse der  $G$ -Algebren ist, folgt die Kommutativität aus der Homomorphieeigenschaft der Codierungsfunktion  $\delta$ . In [120] wird diese Methode auf eine kleine Beispielsprache mit Zuweisungen, while - Schleifen und Deklarationen angewendet.

Mosses hat in [97] diese Methode verfeinert, indem er unter Verwendung abstrakter Datentypen einen korrekten Compiler aus der Semantikdefinition von  $L$  konstruiert. Jones und Schmidt arbeiten ebenfalls in diesem algebraischen Rahmen und haben in [58] zwei Methoden zur Optimierung des erzeugten Codes beschrieben.

## 6.2 Implementierung nach Übersetzung in kombinatorische Ausdrücke

Aus der kombinatorischen Logik ist bekannt, daß sich jeder  $\lambda$ -Ausdruck in einen gleichbedeutenden Kombinatorausdruck, in dem nur die Kombinatoren  $S$ ,  $K$  und  $I$  vorkommen, übersetzen läßt (s. Curry und Feys [29]). Turner hat in [123] die Idee verfolgt, solche Kombinatorausdrücke möglichst hardwarenah zu implementieren, da es einfacher erscheint, eine kleine Anzahl von bestimmten Operationen zu realisieren als die freie Verwendung von Abstraktion und Applikation. Hat man eine Maschine zur Auswertung der Kombinatoren  $S$ ,  $K$  und  $I$  gegeben, so läßt sich ein Programm, dessen Semantik denotationell beschrieben ist, nach Übersetzung des entsprechenden  $\lambda$ -Ausdruckes in einen Kombinatorausdruck darauf ausführen. Natürlich ist eine solche Maschine auch dazu geeignet, funktionale Programme nach Übersetzung in Kombinatorausdrücke auszuwerten. Man kann also in diesem Sinne die systematische Übersetzung von  $\lambda$ -Ausdrücken in kombinatorische Ausdrücke als Compilierung verstehen und das Ergebnis einer solchen Übersetzung als Maschinenprogramm, wenn auch nicht für konventionelle Maschinen.

Die Grundidee bei der Übersetzung von  $\lambda$ -Ausdrücken in Kombinatorausdrücke

ist die Elimination gebundener Variablen. Eine erste Form der Variablenelimination wurde bereits bei der Entwicklung der Standardsemantik häufig verwendet, nämlich die nach dem Gesetz der Extensionalität mögliche Transformation einer Gleichung der Form

$$fx = gx$$

in die Form

$$f = g$$

(vgl. 3.26.4). I.a. hat man jedoch Funktionsgleichungen der Art

$$fx = E ,$$

wobei  $x$  in  $E$  beliebig (also nicht nur einmal ganz am Ende) vorkommen kann. Man sucht also nach einer Transformation  $[x]$  mit der Eigenschaft:  $x$  kommt in  $[x]E$  nicht frei vor und

$$[x]Ex = E .$$

Damit lässt sich dann die Gleichung

$$fx = E$$

in

$$f = [x]E$$

semantikerhaltend transformieren.

Die drei Kombinatoren

$$S f g x = f x (g x)$$

$$K x y = x$$

$$I x = x$$

erlauben eine induktive Definition der Transformation  $[x]$  für applikative Ausdrücke  $E$ :

$$[x]x = I$$

$$[x]y = K y, y \neq x$$

$$[x](E_1 E_2) = S ([x]E_1)([x]E_2) .$$

**Lemma 6.2** Sei  $x \in X$  und  $E$  ein applikativer Ausdruck.

Es gilt:

$$[x]E x = E .$$

Beweis per Induktion über den Aufbau von  $E$ :

$$\begin{aligned}
 \underline{E = x} \quad [x]E x &= I x = x = E \\
 \underline{E = y \neq x} \quad [x]E x &= K y x = y = E \\
 \underline{E = E_1 E_2} \quad [x]E x &= S([x]E_1)([x]E_2) x \\
 &= [x]E_1 x ([x]E_2 x) \\
 &= E_1 E_2 \text{ per Induktionsannahme} \\
 &= E
 \end{aligned}$$

Q.E.D.

Dies ist bereits der erforderliche Algorithmus. Da die Semantikfunktionen in der Regel durch Gleichungen der Form

$$[\ ] x_1 \dots x_r = E$$

gegeben sind, wobei  $E$  ein applikativer Ausdruck ist, genügt es, die o.a. Transformation für alle Variablen  $x_1, \dots, x_r$ , durchzuführen. Da bei diesen Transformationen sehr große Ausdrücke über den Kombinatoren  $S$ ,  $K$  und  $I$  und den Basisfunktionen entstehen, führt Turner in [123] noch eine Optimierung unter Verwendung der Kombinatoren

$$B f g x = f (g x)$$

und

$$C f g x = f x g$$

ein:

1.  $S(K E_1)(K E_2) \implies K(E_1 E_2)$
2.  $S(K E) I \implies E$
3.  $S(K E_1) E_2 \implies B E_1 E_2$
4.  $S E_1(K E_2) \implies C E_1 E_2$ ,

wobei Regel 3. nur dann angewendet wird, wenn Regeln 1. und 2. nicht anwendbar sind und entsprechend die Regel 4. nur dann, wenn Regeln 1. bis 3. nicht anwendbar sind.

Die Korrektheit dieser Optimierung lässt sich leicht durch Anwendung der entsprechenden Ausdrücke auf ein symbolisches Argument nachweisen.

**Beispiel:** Die Fakultätsfunktion ist durch folgende Gleichung definiert:

$$\underline{fac\ n = cond\ (eq\ n\ 0)\ 1\ (times\ n\ (fac\ (minus\ n\ 1)))} .$$

Der Ausdruck  $[n]fac$  lautet:

$$\begin{aligned}
 &S(S(K \underline{cond})(S(S(K \underline{eq})(K 0))I))(K 1)) \\
 &(S(S(K \underline{times})I)(S(K \underline{fac})(S(S(K \underline{minus})I)(K 1)))) .
 \end{aligned}$$

Nach der o.a. Optimierung reduziert er sich zu:

$$S(C(B \text{ cond } (\underline{eq} \ 0))1)(S \text{ times } (B \text{ fac } (C \text{ minus } 1))).$$

Wie man sieht, entsteht ein Code, der unerheblich länger ist als der Quelltext.

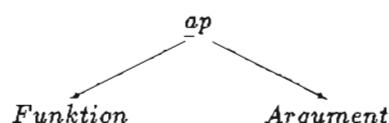
Die von Turner entworfene abstrakte Maschine zur Auswertung dieser kombinatorischen Ausdrücke legt den auszuwertenden Ausdruck in einem Kellerspeicher ab und arbeitet stets auf der Kellerspitze gemäß der Definition der Kombinatoren und der Basisoperationen. Eine ausführliche Beschreibung der Arbeitsweise findet man ebenfalls bei Turner [123].

## 6.3 Implementierung auf Reduktionsmaschinen

In diesem Abschnitt wird wieder die Idee verfolgt, einen funktionalen Ausdruck bzw. den semantischen Ausdruck eines beliebigen Programms durch aufeinanderfolgende Reduktionen in die Darstellung seines Wertes zu überführen. Wie bereits in Kapitel 4 Abschnitt 4 erwähnt, wurde ein solches System (SIS) von Mosses [96] in Aarhus entwickelt. Es existieren auch eine Reihe anderer  $\lambda$ -Kalkül Interpretierer, die sich für diesen Zweck eignen. Auf konventionellen Maschinen sind jedoch alle diese Systeme sehr ineffizient, so daß sie sich nur für Studienzwecke eignen. Hier sollen nun drei innovative Rechnerarchitekturen skizziert werden, die es gestatten, funktionale Ausdrücke effizient zu reduzieren. Die entscheidenden Vorteile gegenüber einer Implementierung durch Compilation bestehen darin, daß eine Einzelschrittausführung möglich ist, die dem Benutzer eine gute Programmierhilfe bietet, und daß die Verifikation der Implementierung einfach ist.

### 6.3.1 Die Reduktionsmaschine von Berkling und Kluge

Die funktionalen Ausdrücke werden intern als Baumstrukturen mit dem binären Applikationsoperator *ap* dargestellt.



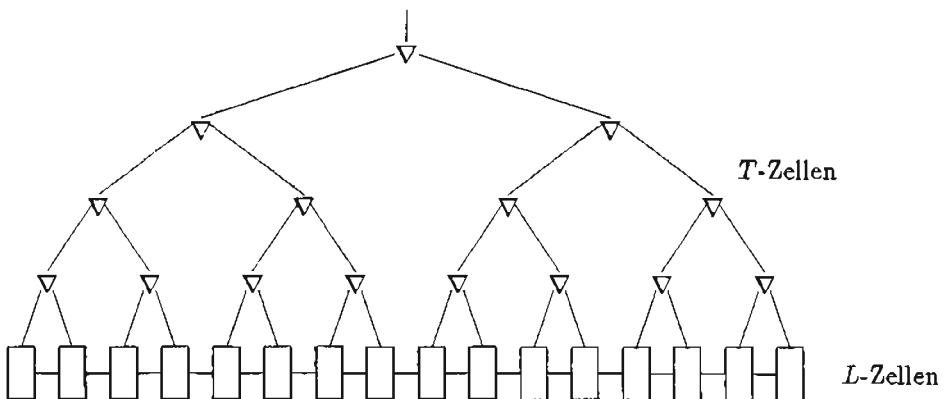
Diese Ausdrücke werden in Präfixnotation in einem Kellerspeicher abgelegt. Die Hauptfähigkeit des Rechners ist ein Traversierungsalgorithmus, der Ausdrücke auf reduzierbare Teilausdrücke untersucht und diese reduziert. Dieser Algorithmus wird auf einem System von Kellerspeichern realisiert, wobei ausschließlich eine Untersuchung der Kellerspitzen die Entscheidung erlaubt, ob reduziert wird oder nicht. Diese Implementierungsart eignet sich in besonderem Maße für die

'call-by-value' - Auswertungsstrategie, die jedoch nicht vollständig ist. Der bei der ersten Realisierung noch vorhandene Nachteil des aufwendigen Transportes großer Teilausdrücke von einem Keller in den anderen, wurde durch die Verwendung eines Hintergrundspeichers aufgehoben. Zeiger auf Teilausdrücke im Hintergrundspeicher werden zunächst wie atomare Objekte behandelt. Steht ein solcher Zeiger zur Ausführung an, wird er in den Kellerspeicher eingelesen und reduziert. Anschließend wird der reduzierte Ausdruck in den Hintergrundspeicher zurückgeschrieben, um eine wiederholte Auswertung zu vermeiden. (Mehrere Zeiger können auf denselben Ausdruck verweisen.)

Ein Hardwaremodell der Reduktionsmaschine wurde 1979 von Berkling und Kluge in der Gesellschaft für Mathematik und Datenverarbeitung erstellt und seither für Testzwecke erfolgreich verwendet (s. [13], [14] und [63]). Eine verteilte Version der Reduktionsmaschine läuft auf einem UNIMA-Rechner mit acht Prozessoren als Emulation (s. [62] und [142]). Ein Nachfolgemodell unter Verwendung der VLSI-Technologie wird konzipiert.

### 6.3.2 Die Baumarchitektur von Mago

Diese Architektur wurde speziell zur Auswertung von FP-Systemen (vgl. Kap. 5 Abschnitt 2) von Mago [79] entwickelt, ließe sich aber auch für eine allgemeinere Klasse funktionaler Ausdrücke erweitern. Der Kern des Rechners besteht aus einem Netz von autonomen Mikroprozessoren, die Zellen genannt werden. Eine Menge  $L$  von linear angeordneten Zellen nimmt den FP-Text in interner Darstellung auf, pro Zelle ein Symbol mit dazugehöriger Schachtelungstiefe. Diese  $L$ -Zellen sind durch eine darüberliegende Baumstruktur von sogenannten  $T$ -Zellen miteinander verbunden. Die  $T$ -Zellen übernehmen Organisationsaufgaben, Berechnungen und Datentransporte. Die Verbindungen zwischen den Zellen stellen Transportkanäle für die Kommunikation zwischen den Zellen dar.



Zur Abarbeitung eines FP-Ausdruckes wird das Netzwerk so aufgeteilt, daß über jedem reduzierbaren Teilausdruck in  $L$  ein unabhängiger Teilbaum von  $T$ -Zellen

entsteht, der die entsprechende Reduktion ausführt.

Das Problem bei dieser Realisierung besteht darin, daß die Länge eines FP-Ausdruckes sich während der Auswertung ständig verändert. Ein reduzierter Teilausdruck kann sowohl kürzer als auch länger als der entsprechende Redex sein. Daher wird der Anfangsausdruck so über die  $L$ -Zellen verteilt, daß zwischen je zwei Symbolen eine möglichst große Anzahl freier Zellen bleibt, die während der folgenden Reduktionen als Puffer verwendet werden können. Somit wird ein häufiges Verschieben der Inhalte der  $L$ -Zellen vermieden. Praktische Erfahrungen mit diesem Maschinenmodell stehen jedoch noch aus.

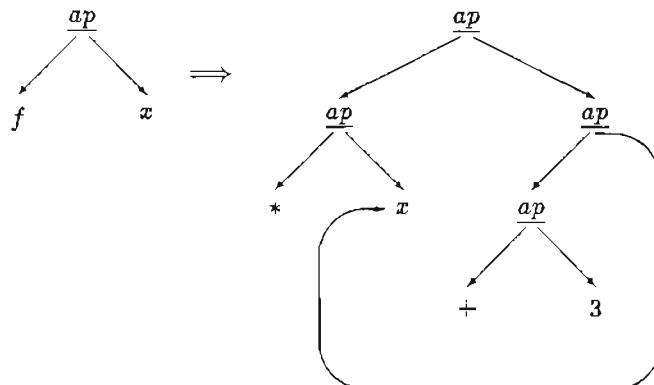
### 6.3.3 Die Graphreduktion nach Wadsworth

Bei dieser Implementierungsart werden die funktionalen Ausdrücke zunächst wieder als Binäräbäume mit dem Konstruktör  $ap$  für die Applikation dargestellt. Nun erlaubt man mehrere Verweise auf einen Teilausdruck. Dadurch entstehen aus den Binäräbäumen sogenannte *dag's* (directed acyclic graphs). Die Reduktionen werden direkt auf der graphischen Darstellung definiert. Anstatt ein Argument mehrmals in einen Funktionsausdruck hineinzukopieren, werden nur die entsprechenden Zeiger auf das Argument gesetzt.

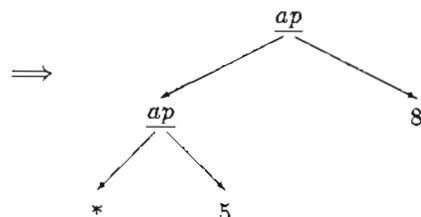
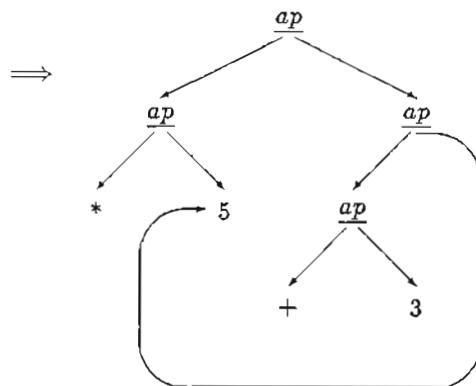
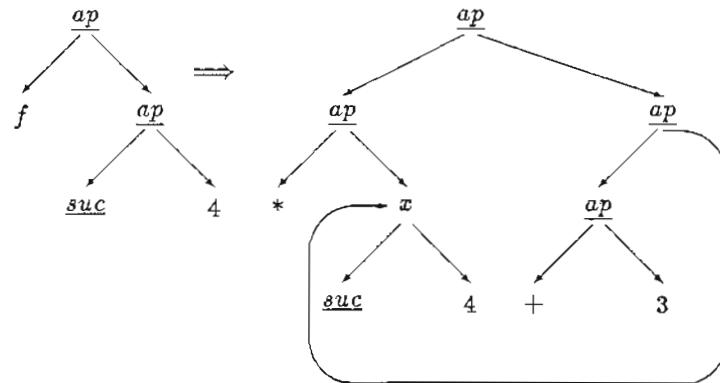
**Beispiel:** Der funktionalen Gleichung

$$f\ x = x * (3 + x)$$

entspricht die folgende Graphtransformation:



Der Aufruf  $f(suc\ 4)$  führt zu folgenden Reduktionen:



$\Rightarrow 40$

Der Vorteil bei Maschinenarchitekturen, die diese Darstellung zugrunde legen, besteht darin, daß eine korrekte Auswertungsstrategie ohne ein Kopieren von Teilausdrücken realisiert werden kann.

Die Graphreduktion wurde erstmalig von Wadsworth in [128] vorgestellt. Darauf aufbauend entwickelten Johnsson und Augustsson die G-Maschine in Göteborg (s. [56] und [6]). Eine gute Beschreibung der Implementierung funktionaler Sprachen mittels Graphreduktion findet man in dem Buch von Peyton Jones [101].

Kersjes stellt in [60] eine Abstrakte Parallel Maschine PAM vor, die ein System von Mikroprozessoren virtuell so organisiert, daß die Kommunikationsstruktur unter den Prozessoren gerade der graphischen Struktur des auszuwertenden Ausdruckes entspricht. Indermark u.a. implementieren diese PAM in OCCAM auf einem Transputersystem [53].

## Literaturverzeichnis

- [1] Aho, A.V. und J.D. Ullman: *The Theory of Parsing, Translation and Compiling*, Vol.I: Parsing, Prentice-Hall (1972)
- [2] Allen, J.: *Anatomy of LISP*, McGraw-Hill (1978)
- [3] Apt, K.R.: *Ten Years of Hoare's Logic: A Survey - Part I*, ACM Transactions of Programming Languages and Systems 3,4 (1981), 431-483
- [4] Arbib, M.A. und E.G. Manes: *Algebraic Approaches to Program Semantics*, Springer-Verlag (1986)
- [5] Aubin, R.: *Mechanizing Structural Induction (Part I)*, Theoretical Computer Science 9,3 (1979), 329-345
- [6] Augustsson, L.: *A Compiler for Lazy ML*, Proc. ACM Symp. on LISP and Functional Programming, Austin (1984), 218-227
- [7] Babich, W.A. und M. Jazayeri: *The Method of Attributes for Data Flow Analysis*, Acta Informatica 10 (1978), 245-272
- [8] Backus, J.: *The Syntax and Semantics of a Proposed International Algebraic Language of the Zürich ACM-GAMM Conference*, Proceedings International Conference Information Processing, UNESCO, Paris (1959), 125-132
- [9] Backus, J.: *Can Programming be Liberated from the von Neumann Style? A functional Style and its Algebra of Programs*, Communications of the ACM 21,8 (1978), 613-641
- [10] De Bakker, J.: *Mathematical Theory of Program Correctness*, Prentice-Hall (1980)
- [11] Barendregt, H.P.: *The Lambda Calculus - Its Syntax and Semantics*, Studies in Logic and the Foundations of Mathematics, vol. 103, North-Holland (1981)
- [12] Bauer, F.L. und H. Wössner: *Algorithmische Sprache und Programmierung*, 2. Auflage, Springer-Verlag (1984)

- [13] Berkling, K.: *Reduction Languages for Reduction Machines*, Proc. 2nd Annual Symposium on Computer Architecture, ACM/IEEE (1975), 133-140
- [14] Berkling, K.: *Computer Architecture for Correct Programming*, Proc. 5th Annual Symposium on Computer Architecture, ACM/IEEE (1978), 78-84
- [15] Bjørner, D.: *Programming Languages: Formal Development of Interpreters and Compilers*, ID673, Dept. of Comp. Science, Tech. Univ. of Denmark, Lyngby (1977)
- [16] Bjørner, D. und C.B. Jones (Hrsg.): *The Vienna Development Method: The Meta-Language*, Lecture Notes in Computer Science No. 61, Springer-Verlag (1978)
- [17] Bjørner, D. und O.N. Oest (Hrsg.): *Towards a Formal Description of Ada*, Lecture Notes in Computer Science No. 98, Springer-Verlag (1980)
- [18] Brauer, W., W. Reisig und G. Rozenberg (Hrsg.): *Petri Nets: Central Models and their Properties*, Proceedings of an Advanced Course Bad Honnef, September 1986, Lecture Notes in Computer Science No. 254 und 255, Springer-Verlag (1987)
- [19] De Bruin, A.: *Goto Statements: Semantics and Deduction Systems*, Acta Informatica 15 (1981), 385-424
- [20] Burstall, R.M.: *Proving Properties of Programs by Structural Induction*, Computer Journal 12 (1969), 41-48
- [21] Burstall, R.M.: *Program Proving as Hand Simulation with a Little Induction*, Proceedings IFIP Congress 74, North-Holland (1974), 308-312
- [22] Burstall, R.M., D.B. MacQueen und D.T. Sanella: *HOPE - An Experimental Applicative Language*, Proceedings LISP-Conference, Stanford (1980), 136-143
- [23] Carnap, R.: *Introduction to Semantics*, Harvard University Press, Cambridge (1942)
- [24] Carnap, R.: *Meaning and Necessity, A Study in Semantics and Modal Logic*, University of Chicago Press, Chicago (1947, erweiterte Auflage 1956)
- [25] Church, A.: *The Calculi of Lambda Conversion*, Princeton University Press (1941)
- [26] Cleveland, J.C. und R.C. Uzgalis: *Grammars for Programming Languages*, Elsevier Science Publ. , New York (1977)
- [27] Cook, S.A.: *Soundness and Completeness of an Axiom System for Program Verification*, SIAM J. Comput. 7,1 (1978), 70-90

- [28] Crowe, D.: *Generating Parsers for Affix Grammars*, Communications of the ACM 15,8 (1972), 728-732
- [29] Curry, H.B., R. Feys und W. Craig: *Combinatory Logic, Vol. I*, North-Holland (1958)
- [30] Damas, L. und R. Milner: *Principal Type Schemes for Functional Programs*, Proc. 9th ACM Symposium on Principles of Programming Languages (1982), 207-212
- [31] Dijkstra E.W.: *A Discipline of Programming*, Prentice-Hall (1976)
- [32] Donahue, J.E.: *Complementary Definitions of Programming Language Constructs*, Lecture Notes in Computer Science No. 42, Springer-Verlag (1976)
- [33] Eick, A. und E. Fehr: *Inconsistencies of pure LISP*, 6th GI-Conference on Theoretical Computer Science in Dortmund, Lecture Notes in Computer Science No. 145, Springer-Verlag (1983), 101-110
- [34] Floyd, R.W.: *Assigning Meanings to Programs*, Proceedings AMS Symposium on Applied Mathematics, Vol.19, American Mathematical Society, Providence R.I. (1967), 19-31
- [35] Frege, G.: *Über Sinn und Bedeutung*, Zeitschrift für Philosophie und philosophische Kritik, 100 (1892), 25-50
- [36] Ganzinger, H.: *Some Principles for the Development of Compiler Description from Denotational Language Definitions*, TUM-INFO der Tech. Univ. München (1979)
- [37] Ganzinger, H.: *Transforming Denotational Semantics into Practical Attribute Grammars*, Semantics-Directed Compiler Generation, Proceedings of a Workshop in Aarhus, Denmark (Hrsg. N.D. Jones), Lecture Notes in Computer Science No. 94, Springer-Verlag (1980), 1-69
- [38] Goguen, J.A., J.W. Thatcher, E.G. Wagner und J.B. Wright: *Initial Algebra Semantics and Continuous Algebras*, Journal of the ACM 24,1 (1977), 68-95
- [39] Gordon, M.J.C.: *The Denotational Description of Programming Languages - An Introduction*, Springer-Verlag (1979)
- [40] Gordon, M.J.C.: *Operational Reasoning and Denotational Semantics*, Stanford University, Computer Science Department, Memo AIM-264 (1975). Auch in: Huet,G. und G.Kahn (Hrsg.): Construction, Amélioration et Vérification des Programmes, Colloques IRIA (1975), 83-98
- [41] Greibach, S. A.: *Theory of Program Structures: Schemes, Semantics, Verification*, Lecture Notes in Computer Science No. 36, (1975)
- [42] Guessarian, I.: *Algebraic Semantics*, Lecture Notes in Computer Science No. 99, Springer-Verlag (1981)

- [43] Halmos, P.R.: *Naive Set Theory*, Springer-Verlag (1960)
- [44] Henderson, P.: *Functional Programming*, Prentice-Hall (1980)
- [45] Henderson, P. und J.M. Morris: *A Lazy Evaluator*, Proceedings of the 3rd Symposium on Principles of Programming Languages (1976), 95-103
- [46] Hesse, W.: *Vollständige formale Beschreibung von Programmiersprachen mit zweischichtigen Grammatiken*, Bericht 7623, Technische Universität München (1976)
- [47] Hindley, J.R., B. Lercher und J.P. Seldin: *Introduction to Combinatory Logic*, Cambridge University Press, London (1972)
- [48] Hoare, C.A.R.: *An Axiomatic Basis for Computer Programming*, Communications ACM 12,10 (1969), 576-580, 583
- [49] Hoare, C.A.R.: *Proofs of Correctness of Data Representations*, Acta Informatica 1,4 (1972), 271-281
- [50] Hoare, C.A.R.: *Communicating Sequential Processes*, Communications of the ACM 21,8 (1978), 666-677
- [51] Hoare, C.A.R. und P.E. Lauer: *Consistent and Complementary Formal Theories of the Semantics of Programming Languages*, Acta Informatica 3 (1974), 135-153
- [52] Hoare, C.A.R. und N. Wirth: *An Axiomatic Definition of The Programming Language PASCAL*, Acta Informatica 2 (1973), 335-355
- [53] Indermark, K., H. Kuchen, R. Loogen und A. Maassen: *Parallele Implementierung einer funktionalen Sprache*, Workshop Sprachen, Algorithmen und Architekturen für Parallelrechner, Bad-Honnef (1988)
- [54] Irons, E.T.: *A Syntax Directed Compiler for ALGOL 60*, Communications of the ACM 4,1 (1961), 51-55
- [55] Jensen, K. und N. Wirth: *PASCAL User Manual and Report*, second edition, Springer-Verlag (1975)
- [56] Johnsson, T.: *Efficient Compilation of Lazy Evaluation*, Proc. ACM Conference on Compiler Construction, Montreal (1984), 58-69
- [57] Jones, C.B.: *Formal Definition in Compiler Development*, TR 25.145, IBM Laboratorium Wien (1976)
- [58] Jones, N.D. und D.A. Schmidt: *Compiler Generation from Denotational Semantics*, Semantics-Directed Compiler Generation, Proceedings of a Workshop in Aarhus, Denmark (Hrsg. N.D. Jones), Lecture Notes in Computer Science No. 94, Springer-Verlag (1980), 70-93

- [59] Kerner, I. (Hrsg.): *Revidierter Bericht über die algorithmische Sprache ALGOL 68*, Akademie-Verlag, Berlin (1978) (deutsche Übersetzung von [138])
- [60] Kersjes, W.H.: *Eine abstrakte Maschine zur optimalen Auswertung funktionaler Programme*, Schriften zur Informatik Nr. 104, RWTH Aachen (1985)
- [61] Kleene, S.C.: *Introduction to Metamathematics*, North-Holland (1952)
- [62] Kluge, W.E. *Cooperating Reduction Machines*, IEEE Transactions on Computers Vol. C-32 No. 11 (1983), 1002-1012
- [63] Kluge, W.E. und H. Schlüter: *An Architecture for Direct Execution of Reduction Languages*, Proc. Int. Workshop on High-Level Language Computer Architecture, Fort Lauderdale, Florida (1980), 174-180
- [64] Knuth, D.E.: *Semantics of Context-free Languages*, Mathematical Systems Theory 2 (1968), 127-145. Korrektur in Mathematical Systems Theory 5 (1971), 95-96
- [65] Knuth, D.E.: *The TeXbook*, Addison-Wesley (1984)
- [66] Koster, C.H.A.: *Affix Grammars*, in J.E.L. Peck (Hrsg.): ALGOL 68 Implementation, North-Holland (1971), 95-109
- [67] Landin, P.J.: *The Mechanical Evaluation of Expressions*, Computer Journal 6,4 (1964), 308-320
- [68] Landin, P.J.: *A Formal Description of ALGOL 60*, Formal Language Description Languages for Computer Programming, Proceedings of the IFIP Working Conference on Formal Language Description Languages, North-Holland (1966), 266-294
- [69] Langmaack, H.: *On Correct Procedure Parameter Transmission in Higher Programming Languages*, Acta Informatica 2 (1973), 110-142
- [70] Langmaack, H.: *On Procedures as Open Subroutines I und II*, Acta Informatica 2 bzw. 3 (1973 bzw. 1974), 311-333 bzw. 227-241
- [71] Lamport, L.: *A Document Preparation System LATEX User's Guide & Reference Manual*, Addison-Wesley (1986)
- [72] Ledgard, H.F.: *Can We Do Better than BNF?*, Communications of the ACM 17,2 (1974), 94-102
- [73] Ledgard, H.F.: *Production Systems: A Notation for Defining Syntax and Translation*, IEEE Transactions on Software Engineering, SE-3, no. 2 (1977), 105-124
- [74] Lewis, P.M., D.J. Rosenkrantz und R.E. Stearns: *Attributed Translations*, Journal of Computer and System Sciences 9 (1974), 279-307

- [75] Lippe, W.M. und F. Simon: *Applikative Programmierung*, Springer-Verlag (1989)
- [76] Loeckx, J. und K. Sieber: *The Foundations of Program Verification*, B.G. Teubner, Stuttgart (1984)
- [77] London, R.L., J.V. Guttag, J.J. Horning, B.W. Lampson, J.G. Mitchel und G.J. Popek: *Proof Rules for the Programming Language EUCLID*, Acta Informatica 10,1 (1978), 1-26
- [78] Lucas, P. und K. Walk: *On the Formal Definition of PL/I*, in M. Halpern und C. Shaw (Hrsg.): Annual Review in Automatic Programming 6, Pergamon Press (1971), 105-182
- [79] Mago, G.A.: *A Network of Microprocessors to Execute Reduction Languages*, International Journal of Computer and Information Sciences Vol. 8, No. 5 (1979), 435-471
- [80] Manna, Z.: *Mathematical Theory of Computation*, McGraw-Hill computer science series (1974)
- [81] Manna, Z., S. Ness und J. Vuillemin: *Inductive Methods for Proving Properties of Programs*, Communications of the ACM 16,8 (1973), 491-502
- [82] Marcotty, M., H.F. Ledgard und G.V. Bochmann: *A Sampler of Formal Definitions*, ACM Computing Surveys 8,2 (1976), 191-276
- [83] McCarthy, J.: *Recursive Functions of Symbolic Expressions and their Computation by Machine*, Communications of the ACM 3,4 (1960), 184-195
- [84] McCarthy, J.: *Towards a Mathematical Science of Computation*, in C.M. Popplewell (Hrsg.): Proc. IFIP Congress 62, North-Holland (1963) 21-28
- [85] McCarthy, J.: *A Basis for a Mathematical Theory of Computation*, in Bradford, P. und D. Hirschberg (Hrsg.): Computer Programming and Formal Systems, North-Holland (1963), 33-70
- [86] McCarthy, J., R. Brayton, D. Edwards, P. Fox, L. Hodes, D. Luckham, K. Maling, D. Park und S. Russel: *LISP 1 Programmer's Manual*, Artificial Intelligence Group, Computation Center and Research Laboratory of Electronics, MIT-Press, Cambridge, Massachusetts (1960)
- [87] McCarthy, J., P.W. Abrahams, D.J. Edwards, T.P. Hart und M.I. Levin: *LISP 1.5 Programmer's Manual*, MIT-Press, Cambridge, Massachusetts (1960)
- [88] McGowan, C.L.: *An Inductive Proof Technique for Interpreter Equivalence*, Courant Computer Science Symposium 2, 1970, in Rustin, R.(Hrsg.): *Formal Semantics of Programming Languages*, Prentice-Hall (1972), 139-147

- [89] McGowan, C.L.: *The Correctness of a Modified SECD Machine*, Second ACM-Symposium on Theory of Computing (1970), 149-157
- [90] Milne, R. und C. Strachey: *A Theory of Programming Language Semantics, Part a and b*, Chapman and Hall (1976)
- [91] Milner, R.: *Implementation and Applications of Scott's Logic for Computable Functions*, Proceedings ACM Conference on Proving Assertions about Programs, Las Cruces, New Mexico (1972)
- [92] Milner, R.: *A Calculus of Communicating Systems*, Lecture Notes in Computer Science No. 92, Springer-Verlag (1980)
- [93] Milner, R.: *A Proposal for Standard ML*, ACM-Symposium on LISP and Functional Programming, Austin, Texas (1984), 184-197
- [94] Morris, F.L.: *Advice on Structuring Compilers and Proving them Correct*, 1st ACM Symposium on Principles of Programming Languages, Boston (1973), 144-152
- [95] Mosses, P.D.: *Compiler Generation Using Denotational Semantics*, Mathematical Foundations of Computer Science in Gdansk, Lecture Notes in Computer Science No. 45, Springer-Verlag (1976), 436-441
- [96] Mosses, P.D.: *SIS - Semantics Implementation System (Reference Manual and User Guide)*, DIAMI MD - 30, Computer Science Dept., Aarhus University (1979)
- [97] Mosses, P.D.: *A Constructive Approach to Compiler Correctness*, in Proceedings of the Seventh International Colloquium on Automata, Languages and Programming, Noordwijkerhout (Hrsg. J. de Bakker und J. van Leeuwen), Lecture Notes in Computer Science No. 85, Springer-Verlag (1980)
- [98] Neel, D. und M. Amirchahy: *Semantic Attributes and Improvement of Generated Code*, Proceedings ACM 1974 Annual Conference (1974), 1-10
- [99] Pagan, F.G.: *Formal Specification of Programming Languages: A Panoramic Primer*, Prentice-Hall (1981)
- [100] Park, D.: *Fixpoint Induction and Proofs of Program Properties*, in B. Meltzer und D. Michie (Hrsg.): Machine Intelligence 5, Edinburgh University Press (1969), 59-78
- [101] Peyton Jones, S.L.: *The Implementation of Functional Programming Languages*, Prentice-Hall (1987)
- [102] Plotkin, G.D.: *Call-by-Name, Call-by-Value and the  $\lambda$ -Calculus*, Theoretical Computer Science 1, North-Holland (1975), 125-159

- [103] Räihä, K.-J.: *On Attribute Grammars and their Use in a Compiler Writing System*, Report A-1977-4, Dept. of Computer Science, University of Helsinki (1977)
- [104] Raskovsky, M. und P. Collier: *From Standard to Implementation Denotational Semantics*, in Semantics-Directed Compiler Generation, Proceedings of a Workshop in Aarhus, Denmark (Hrsg. N.D. Jones), Lecture Notes in Computer Science No. 94, Springer-Verlag (1980), 94-139
- [105] Reisig, W.: *Petrinetze - Eine Einführung*, 2. Auflage, Springer-Verlag (1986)
- [106] Riedewald, G., J. Maluszynski und P. Dembiński: *Formale Beschreibung von Programmiersprachen*, Oldenbourg (1983)
- [107] Scott, D.S.: *Outline of a Mathematical Theory of Computation*, Proceedings of Fourth Annual Princeton Conference on Information Science and Systems, Princeton (1970), 169-176
- [108] Scott, D.S.: *Continuous Lattices*, Technical Monograph PRG-7, Proceedings of Dalhousie Conference, Lecture Notes in Mathematics No. 274, Springer-Verlag (1972), 97-134
- [109] Scott, D.S.: *Data Types as Lattices*, SIAM Journal on Computing, 5 (1976), 522-587
- [110] Scott, D.S.: *Domains for Denotational Semantics*, Ninth International Colloquium on Automata, Languages and Programming in Aarhus, Lecture Notes in Computer Science No. 140, Springer-Verlag (1982), 577-613
- [111] Scott, D.S. und C. Strachey: *Towards a Mathematical Semantics for Computer Languages*, in Proc. of the Symposium on Computers and Automata (Hrsg. J. Fox), Polytechnic Institute of Brooklyn Press, New York (1971), 19-46; auch: Technical Monograph PRG-6, Programming Research Group, University of Oxford (1971)
- [112] Smyth, M. und G.D. Plotkin: *The Category-Theoretic Solution of Recursive Domain Equations*, SIAM Journal on Computing, Vol.11, No. 4 (1982), 761-783
- [113] Stoy, J.E.: *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT-Press (1977)
- [114] Strachey, C.: *Towards a Formal Semantics*, Formal Language Description Languages for Computer Programming, Proceedings of the IFIP Working Conference on Formal Language Description Languages, North-Holland (1966), 198-220
- [115] Strachey, C. und C.P. Wadsworth: *Continuations - a Mathematical Semantics for Handling Full Jumps*, Technical Monograph PRG-11, Programming Research Group, University of Oxford (1974)

- [116] Tarski, A.: *A Lattice-Theoretical Fixpoint Theorem and its Application*, Pacific Journal of Mathematics 5 (1955), 285-309
- [117] Tarski, A.: *Logic, Semantics, and Meta-mathematics*, Oxford University Press, Oxford (1956)
- [118] Tennent, R.D.: *The Denotational Semantics of Programming Languages*, Communications of the ACM 19 (1976), 437-453
- [119] Tennent, R.D.: *A Denotational Definition of the Programming Language PASCAL*, Technical Report 77-47, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada (1977)
- [120] Thatcher, J.W., E.G. Wagner und J.B. Wright: *More on Advice on Structuring Compilers and Proving them Correct*, in Poceedings of the Sixth International Colloquium on Automata, Languages and Programming, Graz (Hrsg. H.A. Maurer), Lecture Notes in Computer Science No. 71, Springer Verlag (1979), 596-615
- [121] Turner, D.A.: *SASL language Manual*, St. Andrews University (1976)
- [122] Turner, D.A.: *The Future of Applicative Programming*, in Duijvestijn, A.J.W. und P.C. Lockemann (Hrsg.): Trends in Information Processing Systems, Lecture Notes in Computer Science No. 123, Springer Verlag (1981), 334-348
- [123] Turner, D.A.: *A New Implementation Technique for Applicative Languages*, Software - Practice and Experience 9,1 (1979), 31-49
- [124] Turner, D.A.: *Miranda: A non-strict functional language with polymorphic types*, in Functional Programming Languages and Computer Architecture (Hrsg. J.P. Jouannaud), Lecture Notes in Computer Science No. 201, Springer Verlag (1985), 1-16
- [125] Turner, D.A.: *An Overview of Miranda*, SIGPLAN Notices vol. 21, no. 12 (1986), 158-166
- [126] Turner, D.A.: *Miranda System Manual*, Copyright Research Software Limited (1987)
- [127] Uhl, J., S. Drossopoulou, G. Persch, G. Goos, M. Dausmann, G. Winterstein und W. Kirchgässner: *An Attribute Grammar for the Semantic Analysis of Ada*, Lecture Notes in Computer Science No. 139, Springer-Verlag (1982)
- [128] Wadsworth, C.: *Semantics and Pragmatics of the Lambda-Calculus*, Diss. Oxford (1971)
- [129] Waite, D. und G. Goos: *Compiler Construction*, Springer-Verlag (1984)

- [130] Wand, M.: *Deriving Target Code as a Representation of Continuation Semantics*, ACM Transactions on Programming Languages and Systems 4,3 (1982), 496-517
- [131] Watt, D.A.: *An Extended Attribute Grammar for PASCAL*, SIGPLAN Notices Vol. 14, No. 2 (1979), 60-74
- [132] Watt, D.A.: *The Parsing Problem for Affix Grammars*, Acta Informatica 8,1 (1977), 1-20
- [133] Wegner, P.: *Programming Languages, Information Structures and Machine Organization*, McGraw-Hill, New York (1968)
- [134] Wegner, P.: *Programming Language Semantics*, Courant Computer Science Symposium 2, 1970 in Rustin, R.(Hrsg.): *Formal Semantics of Programming Languages*, Prentice-Hall (1972), 149-248
- [135] Wegner, P.: *Operational Semantics of Programming Languages*, ACM Symposium on Proving Assertions about Programs (1972)
- [136] Wegner, P.: *The Vienna Definition Language*, Computing Surveys 4,1 (1972), 5-63
- [137] Wilhelm, R.: *Computation and Use of Data Flow Information in Optimizing Compilers*, Acta Informatica 12 (1979), 209-225
- [138] Van Wijngaarden, A.: *Revised Report on the Algorithmic Language ALGOL 68*, Springer-Verlag (1976)
- [139] Wilner, W.T.: *Declarative Semantic Definition as Illustrated by a Definition of SIMULA 67*, Ph. D. dissertation Stanford University, Stanford, California (1971)
- [140] Wirth, N.: *What Can We Do about the Unnecessary Diversity of Notations for Syntactic Definitions?*, Communications of the ACM 20,11 (1977), 822-823
- [141] Zima, H.: *Compilerbau I und II*, Bibliographisches Institut, Zürich (1982) und (1983)
- [142] Zimmer, R.: *A Heap-Supported Multiprocessor Reduction System*, Workshop Sprachen, Algorithmen und Architekturen für Parallelrechner, Bad-Honnef (1988)

# Studienreihe Informatik

---

Herausgegeben von W. Brauer und G. Goos

- P. C. Lockemann, H. C. Mayr: **Rechnergestützte Informationssysteme.** X, 368 S., 37 Abb. 1978.
- A. K. Salomaa: **Formale Sprachen.** Übersetzt aus dem Englischen von E.-W. Dieterich. IX, 314 S., 18 Abb., 5 Tab. 1978.
- F. L. Nicolet (Hrsg.): **Informatik für Ingenieure.** Unter Mitarbeit von W. Gander, J. Harms, P. Läuchli, F. L. Nicolet, J. Vogel, C. A. Zehnder. X, 187 S., 53 Abb., 20 Tab. 1980.
- A. Bode, W. Händler: **Rechnerarchitektur – Grundlagen und Verfahren.** XI, 278 S., 140 Abb., 4 Tab. 1980.
- B. W. Kernighan, P. L. Plauger: **Programmierwerkzeuge.** Übersetzt aus dem Englischen von I. Kächele, M. Klopprogge. IX, 492 S. 1980.
- A. N. Habermann: **Entwurf von Betriebssystemen – Eine Einführung.** Übersetzt aus dem Englischen von K.-P. Lohr. XII, 444 S., 87 Abb. 1981.
- T. W. Olle: **Das Codasyl-Datenbankmodell.** Übersetzt aus dem Englischen von H. Münzenberger. XXIV, 389 S. 1981.
- K. E. Ganzhorn, K. M. Schulz, W. Walter: **Datenverarbeitungssysteme – Aufbau und Arbeitsweise.** XVI, 305 S., 181 Abb., 1 Schablone als Beilage. 1981.
- B. Buchberger, F. Lichtenberger: **Mathematik für Informatiker I – Die Methode der Mathematik.** 2., korrigierte Auflage. XIII, 315 S., 30 Abb. 1981.
- F. Gebhardt: **Dokumentationssysteme.** 331 S., 14 Abb. 1981.
- E. Horowitz, S. Sahni: **Algorithmen – Entwurf und Analyse.** Übersetzt aus dem Amerikanischen von M. Czerwinski. XIV, 770 S. 1981.
- W. Sammer, H. Schwärtzel: **CHILL – Eine moderne Programmiersprache für die Systemtechnik.** XIII, 191 S., 165 Abb. 1982.
- P. C. Lockemann, A. Schreiner, H. Trauboth, M. Klopprogge: **Systemanalyse – DV-Einsatzplanung.** XIV, 342 S., 119 Abb. 1983.
- A. Bode, W. Händler: **Rechnerarchitektur II – Strukturen.** XI, 328 S., 164 Abb. 1983.
- H. A. Klaeren: **Algebraische Spezifikation – Eine Einführung.** VII, 235 S. 1983.
- H. Niemann: **Klassifikation von Mustern.** X, 340 S., 77 Abb. 1983.
- F. L. Bauer, H. Wössner: **Algorithmische Sprache und Programmentwicklung.** Unter Mitarbeit von H. Partsch, P. Pepper. 2., verbesserte Auflage. XV, 513 S. 1984.
- H. Stoyan, G. Görz: **LISP – Eine Einführung in die Programmierung.** XI, 358 S., 29 Abb. 1984.
- K. Däßler, M. Sommer: **Pascal – Einführung in die Sprache; DIN-Norm 66256; Erläuterungen.** 2. Auflage. Unter Mitarbeit von A. Biedl. XIII, 248 S. 1985.
- G. Blaschek, G. Pomberger, F. Ritzinger: **Einführung in die Programmierung mit Modula-2.** VIII, 279 S., 26 Abb. 1987.

- 
- R. Marty: **Methodik der Programmierung in Pascal.** 3. Auflage. IX, 201 S., 33 vollständige Programmbeispiele. 1986.
- W. Reisig: **Petrinetze – Eine Einführung.** 2., überarbeitete und erweiterte Auflage. IX, 196 S., 111 Abb. 1986.
- J. Nievergelt, K. Hinrichs: **Programmierung und Datenstrukturen – Eine Einführung anhand von Beispielen.** XI, 149 S. 1986.
- E. Jessen, R. Valk: **Rechensysteme – Grundlagen der Modellbildung.** XVI, 562 S., 269 Abb. 1987.
- F. Stetter: **Grundbegriffe der Theoretischen Informatik.** VIII, 236 S., 39 Abb. 1988.
- H. Stoyan: **Programmiermethoden der Künstlichen Intelligenz. Bd. 1.** XV, 343 S. 1988.
- F. Puppe: **Einführung in Expertensysteme.** X, 205 S., 79 Abb. 1988.
- W. Heise, P. Quattrocchi: **Informations- und Codierungstheorie – Mathematische Grundlagen der Daten-Kompression und -Sicherung in diskreten Kommunikationssystemen.** 2., neubearbeitete Auflage. XII, 392 S., 114 Abb. 1989.
- E. Fehr: **Semantik von Programmiersprachen.** IX, 202 S., 1989.

## **Fehr Semantik von Programmiersprachen**

Dieses Buch vermittelt Techniken zur Formalisierung der Semantik von Programmiersprachen. Zunächst werden unterschiedliche Formalisierungsansätze (operationelle, denotationelle und axiomatische Semantik) vorgestellt und diskutiert. Anschließend wird die mathematische Theorie der semantischen Bereiche entwickelt, die bei der zur Zeit wichtigsten, der denotationellen Methode, Anwendung findet. Danach wird schrittweise eine umfassende, PASCAL-orientierte Programmiersprache entworfen, und die Semantik der einzelnen Sprachelemente wird denotationell spezifiziert. Die Fortsetzungssemantik (continuation semantics) wird dabei systematisch erklärt und verwendet. Schließlich wird die Anwendung dieser Techniken betrachtet, insbesondere im Rahmen des Compilerbaus und als Grundlage zur Entwicklung funktionaler Programmiersprachen.

ISBN 3-540-15163-X  
ISBN 0-387-15163-X